Perguntas e Respostas:
Hint's gráficos e personalizados
Modularize sua aplicação sem modificar uma linha de código

Clube Delphi +PHP

DevMedia group

Ano 9 :: Edição 114 :: R$ 11,90

SPED Contábil e Fiscal
Entenda como fazer a geração do arquivo solicitado pelo SPED

Boas Práticas
Abstract Factory:
Crie objetos da forma correta

Quick Update
JSON no DataSnap 2010

Design Patterns

Crie um sistema de Logs usando excelentes práticas de programação OO

Brinde

Comprando esta edição ganhe 4 vídeo-aulas sobre Rave Reports

- Conheça os componentes RvProject e RvSystem
- Inserindo imagens dinamicamente
- Exportando relatórios
- Visões, Connections e Security Controller

Entendendo Generics
O que você precisa saber para começar a usar Generics em seus projetos Delphi

PHP
Migre passo a passo uma aplicação OO para usar o poderoso padrão MVC

ISSN 1517990-7

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.

Sumário

Quick Update

8 –JSON no DataSnap 2010
[*Rodrigo Carreiro Mourão*]

Delphi — Mão na Massa

11 – SPED Contábil e Fiscal
Entenda como fazer a geração do arquivo solicitado pelo SPED
[*Jederson Zuchi*]

Boas Práticas — Boas Práticas

18 – Programação Orientada a Objetos
Utilize Generics, Padrões de Projeto e OO para criar um Log das ações do sistema
[*Rafael Stavarengo*]

Boas Práticas — Boas Práticas

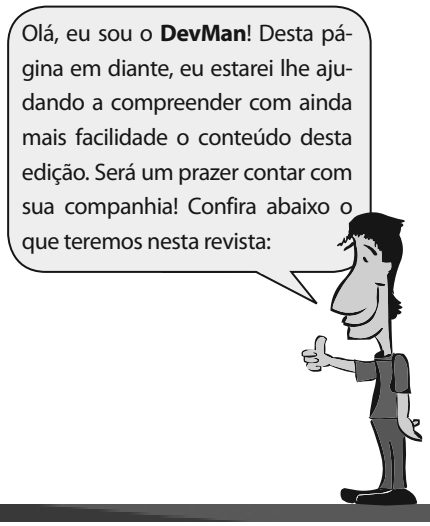
25 – Abstract Factory
Flexibilizando a criação de objetos
[*Giuliano Prado de Morais Giglio e Tadeu Moreira de Classe*]

Easy Delphi — Easy Delphi, Boas Práticas

31 – Entendendo Generics
O que você precisa saber para começar a usar Generics em seus projetos Delphi
[*Vitor Luiz Rubio*]

PHP — PHP, Boas Práticas

40 – MVC com PHP na prática
Migre passo a passo uma aplicação OO para usar o poderoso padrão MVC
[*Lucas Simões Maistro e Raphael Zanon Rodrigues*]



Brinde na web desta edição - 4 vídeos! Confira:

4 Vídeos

- **Rave Reports – RvProject e RvSystem**
  http://www.devmedia.com.br/articles/viewcomp.asp?comp=14430
- **Rave Reports – DataSet, NDRWrite, Render Printer e Render Preview**
  http://www.devmedia.com.br/articles/viewcomp.asp?comp=14431
- **Rave Reports – BMPImage**
  http://www.devmedia.com.br/articles/viewcomp.asp?comp=14432
- **Rave Reports – Gerenciando Dataviews e criando Connections**
  http://www.devmedia.com.br/articles/viewcomp.asp?comp=14455

Para acessar os vídeos utilize os seguintes dados
Login: ***DVM.CD*** *Senha:* ***cloudelphi***

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.CD** Senha: **cloudelphi**

Clube Delphi +PHP

Ano 9 - 114ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editores Técnicos
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.laporte@gmail.com

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Capa
Romulo Araujo
romulo@devmedia.com.br

Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Separe e esconda. A OO diz que devemos encapsular o que muda. Isso fará com que um uma mudança no sistema, por exemplo, em um requisito, na arquitetura da solução, nas tecnologias envolvidas, no banco de dados, no tipo de interface cliente, tenha um impacto bem menor. Separar e definir responsabilidades únicas a cada uma das classes envolvidas no sistema, isso garante um baixo acoplamento, uma fraca dependência. Você desenvolve assim? Faça um pequeno teste. Inicie um novo projeto em Delphi, peça ao IDE para adicionar uma nova unit ao seu projeto vazio e selecione aquelas units com as classes mais genéricas do seu framework, aquelas do tipo "util.pas" que você utiliza em todos os projetos. Verifique se ela irá compilar sem problemas, sem que esteja atrelada (acoplada) a outras classes (tipo units específicas de projetos como Estoque, Financeiro, Acadêmico). Se isso ocorrer, você pode até resolver usando um pouco de POG colocando um IFDEF na sua util.pas e definindo uma diretiva condicional no projeto que a usa (eu já fiz isso! Bom, eu era estagiário...). Aqui na ClubeDelphi trabalhamos com profissionalismo, e como de costume, como o amigo leitor deve ter certamente reparado nas últimas edições, trazemos não só o melhor do Delphi a cada mês, mas que este melhor seja desenvolvido e apresentado corretamente, da melhor forma, usando boas práticas, excelentes práticas. Programação Orientada a Objetos, Design Patterns, Interfaces, Generics, RTTI, Encapsulamento, Polimorfismo, têm sido temas frequentes em nossa revista. Confira a matéria de capa do Rafael e comprove!

E aqui vai outra dica para tornar suas classes menos acopladas e com maior reaproveitamento: tome cuidado com o lugar onde você escreve o TSuaClasse.Create(). Isso porque uma classe não deve conhecer a implementação concreta de outra classe, mas precisa se ligar a ela através de uma interface, ou uma classe mais básica, ou ainda abstrata. Na verdade, quando uma classe A quiser instanciar uma outra do tipo B, deve pedir isso a uma outra classe C (chamada de fábrica). Esse é um "problema" clássico da OO para o qual já existe uma solução, um padrão de projeto criacional, mostrado na prática no artigo do Giuliano e Tadeu.

Cansado de passar ou declarar TObject em procedures que precisam receber qualquer coisa? Ou pior ainda, Pointer? Esqueça. Generics é um ingrediente que estava faltando há anos no Delphi, agora disponível para Win32 (Delphi 2009 / 2010). O Vitor desvenda tudo sobre este tema que já estamos discutindo há algumas edições aqui na ClubeDelphi. Nesta edição, o tema é bastante aprofundado: você verá como criar uma classe genérica para trabalhar com dados fortemente tipados, igualzinho ao que fez o .NET para coleções (e agora o Delphi Win32 também).

Na parte Web, com PHP, também não deixamos as boas práticas de lado! O Raphael e o Lucas mostram como implementar o famoso padrão MVC. Não deixe de conferir também nesta edição uma matéria do Jaderson, sobre a integração do Delphi com SPED.

Boa Leitura.

Guinther Pauli

guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Perguntas e Respostas

Perguntas respondidas por:
Daniel Laporte: daniel.laporte@gmail.com
Guinther Pauli: guinther.pauli@gmail.com

Criando um Hint gráfico personalizado

Estou criando minhas primeiras aplicações com o Delphi 2010 e gostaria de aprimorar a minha interface de usuário, principalmente no que diz respeito à exibição de dicas (hints). Eu realmente não gostaria de usar componentes de terceiros para isso, existe algum recurso ou componente nativo do Delphi que permita a criação de hints mais elaborados? *(Jackson Rubrin - Belo Horizonte - MG)*

Como sabemos, a partir do Delphi 2009, tivemos a adição de um novo componente chamado BallonHint que permite criar mensagens hints mais elegantes através de uma sintaxe bem simples, usando a seguinte estrutura:

Título do Hint | *Mensagem* | *Índice de imagem*

Veja um exemplo na **Figura 1.**

Figura 1. Ballon Hint do Delphi 2009 / 2010

Para simularmos um Hint mais elegante criaremos uma classe de Hint gráfico. O código da classe se encontra na **Listagem 1.**

Para visualizar o resultado crie uma nova aplicação VCL, insira um Edit e informe a unit da classe que criamos no uses do formulário. Coloque um Glyph de sua preferência com o

Listagem 1. Classe de Hint gráfico

```
unit uGraphicHintWindow;

interface

uses Windows, Controls, Classes, Types, Forms, Graphics, SysUtils;

type
  TGraphicHintWindow = class(THintWindow)
    constructor Create(AOwner: TComponent); override;
  private
    FActivating: Boolean;
  public
    procedure ActivateHint(Rect: TRect; const AHint: string);
override;
  protected
    procedure Paint; override;
  published
    property Caption;
  end;

implementation

{ TGraphicHintWindow }

procedure TGraphicHintWindow.ActivateHint(Rect: TRect;
  const AHint: string);
begin
  FActivating := True;
  try
    Caption := AHint;
    Inc(Rect.Bottom, 14);
    Rect.Right := Rect.Right + 20;
    UpdateBoundsRect(Rect);
    if Rect.Top + Height > Screen.DesktopHeight then
      Rect.Top := Screen.DesktopHeight - Height;
    if Rect.Left + Width > Screen.DesktopWidth then
      Rect.Left := Screen.DesktopWidth - Width;
    if Rect.Left < Screen.DesktopLeft then
      Rect.Left := Screen.DesktopLeft;
    if Rect.Bottom < Screen.DesktopTop then
      Rect.Bottom := Screen.DesktopTop;
    SetWindowPos(Handle, HWND_TOPMOST, Rect.Left, Rect.Top, Width,
    Height,
      SWP_SHOWWINDOW or SWP_NOACTIVATE);
    Invalidate;
  finally
    FActivating := False;
  end;
end;

constructor TGraphicHintWindow.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  with Canvas.Font do
  begin
    Name := 'Arial';
    Style := Style + [fsBold];
    Color := clBlack;
  end;
end;
```

Continuação: Listagem 1. Classe de Hint gráfico

```
procedure TGraphicHintWindow.Paint;
var
  R: TRect;
  bmp: TBitmap;
  bFile: String;
begin
  R := ClientRect;
  Inc(R.Left, 2);
  Inc(R.Top, 2);
  bFile := ExtractFilePath(Application.ExeName) + 'hint.bmp';
  bmp := TBitmap.Create;
  if FileExists(bFile) then
    bmp.LoadfromFile(bFile);
  with Canvas do
  begin
    Brush.Style := bsSolid;
    Brush.Color := clsilver;
    Pen.Color   := clgray;
    Rectangle(0, 0, 18, R.Bottom + 1);
    Draw(2,(R.Bottom div 2) - (bmp.Height div 2), bmp);
  end;
  bmp.Free;
  Color := clWhite;
  Canvas.Brush.Style := bsClear;
  Canvas.TextOut(20, (R.Bottom div 2) - (Canvas.
  Textheight(Caption) div 2), Caption);
end;

end.
```

nome de hint.bmp na pasta do projeto e no evento OnCreate faça o seguinte código:

```
HintWindowClass := TGraphicHintWindow;
Application.ShowHint := False;
Application.ShowHint := True;
```

Veja na **Figura 2** como ficou nosso exemplo.

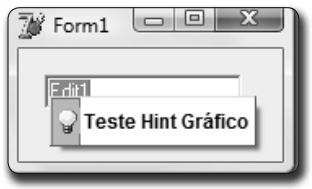


Figura 2. Resultado do Hint Gráfico

Modularização com BPL's – Impacto zero no código

Gostaria de saber qual a melhor forma de modularizar uma aplicação Delphi (Ex.: Clientes.Exe, Financeiro.Exe etc.).
(Wekslei Monteiro da Silva - Rio de Janeiro)

A melhor forma de modularizar sua aplicação é usando BPL's. Se você usar múltiplos EXE's, terá algums problemas que precisarão ser resolvidos manualmente, como por exemplo, compartilhar o LOGIN feito no módulo principal com os demais EXE's. Uma conexão com o BD tamém fica difícil de ser compartilhada entre vários EXE's. A solução que causa menor impacto, inclusive se você já está com o projeto pronto ou em desenvolvimento, é a seguinte:

1 - Crie um Package (menu File>New) para cada módulo do sistema. Exemplo "Financeiro.dpk";

2 - Salve este pacote (e os demais) no mesmo diretório do seu DPR (não é obrigatório, mas facilita);

3 - Coloque dentro de cada DPK os forms e units relativas a cada módulo;

4 - Dê um build em todos os pacotes, o que irá gerar várias BPL's;

5 - Agora vem a melhor parte, no seu DPR, em Project>Options, apenas marque que você deseja usar a opção "Build with runtime packages", não precisa fazer mais nada, nem mesmo retirar os forms que foram para as BPLs, eles permanecem no DPR (ou seja, alteração zero no código);

6 - Ainda nessa opção, adicione o nome dos pacotes na lista (extensão DCP);

7 - Dê um BUILD no DPR;

8 - Vá em Project>Information e veja se o Delphi indica a dependência dos módulos (BPL's), que agora precisam ser distribuídos juntamente com o EXE.

Dica

Use um project group para gerenciar todos os seus módulos e EXE principal.

Se precisar de mais infos, segue link:
http://www.devmedia.com.br/articles/viewcomp.asp?comp=551

Envie suas perguntas para nossa equipe de profissionais e editores: guinther.pauli@gmail.com, daniel.laporte@gmail.com, rodrigocarreiro@rmfactory.com.br, vitorrubio@gmail.com e faelsta@gmail.com. Dúvidas podem ser resumidas / editadas por restrições de espaço. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

local
Data Center

Nossos **servidores** dedicados possuem monitoramento 24x7x365, link dedicado e **full duplex**, ou seja, os servidores atingem a **velocidade** do link tanto para **upload** quanto para **download.**

HOST

Servidor
Sites
Gerenciamento

DATACENTER

Co-location
E-mail
Backup

TELECOM

Cross Connection
Banda IP
Link PAP

SOLUÇÕES INTELIGENTES

Solicite um representante comercial

www.localdatacenter.com.br

SOLICITE UM ORÇAMENTO
TEL.: 3956 - 4900
comercial@localdatacenter.com.br

Quick Update:
JSON no DataSnap 2010

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br
Apaixonado por tecnologia da informação. Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Editor Técnico da Revista ClubeDelphi e Coordenador da área de Delphi no sistema de suporte on-line da DevMedia. Palestrante da Borland Conference 2007. Mantenedor da comunidade www.delphisophp.com, a primeira do Brasil 100% feita com Delphi for PHP. Autor de diversos artigos técnicos veiculados nos principais sites e revistas sobre Delphi do Brasil .

Como todos sabem muitas novidades foram adicionadas nesta nova versão do Delphi, o Delphi 2010. Entre tantas novidades temos agora no novo DataSnap o suporte a JSON. Mas você pode estar se perguntando: "O que é que eu tenho a ver com isso?"

Se algum dia na sua vida você trabalhou com aplicações multicamadas e já quebrou a cabeça para poder trabalhar com OO então isso tem tudo a ver com você. Isso porque o conceito, a essência de uma aplicação multicamadas é justamente concentrar as regras de negócio no servidor de aplicação, assim como em OO as regras de negócio estão em nossas classes então nossas classes deveriam estar no servidor, correto? Mas aí vem a grande questão: como criar o objeto no servidor e ter um "espelho" dele no cliente ou como tê-lo "fisicamente" no cliente?

Os mais entendidos diriam: Usamos COM para isso. Mas o COM já foi descontinuado e era um recurso não muito fácil de trabalhar. E se eu lhe dissesse que agora pode transportar seu objeto do cliente para o servidor e vice e versa de uma forma simples, rápida e o melhor, com um desempenho impressionante? E se além de fazer isso entre um servidor de aplicação Delphi e um cliente Delphi eu pudesse fazer isso entre um servidor Delphi e uma aplicação cliente feita em uma das mais de 20 linguagens que suporte JSON hoje em dia? Aí fica melhor ainda.

Imagine você que seu servidor de aplicação, feito em Delphi, está sendo usado por diversas empresas, de diferentes segmentos. Obviamente cada uma dessas empresas possui sua metodologia de desenvolvimento e como

tal cada uma utiliza uma linguagem específica, seja PHP, C#, Java, Ruby etc. e todas elas estarão "falando" com seu servidor de aplicação, feito em Delphi, de forma transparente. Neste ponto você poderá dizer: "Interessante, mas o que vem a ser o JSON?"

O JSON (JavaScript Object Notation) é uma espécie de padronização para transporte de informações entre ambientes heterogêneos e que tem como linguagem o Javascript. Bem, padrão, transporte, ambiente heterogêneo, tudo isso nos remete ao XML, então vem a questão: Por que JSON e não XML? O principal motivo seria pelo fato da notação JSON ser extremamente mais simples do que a XML. Não existem tags diversas nem regras complexas para escrita e muito menos mecanismos complexos de parser. Para se ter uma ideia observe a **Figura 1**, ela mostra a sintaxe para notação JSON para um objeto.

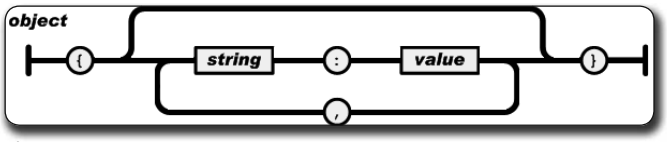


Figura 1. Sintaxe para notação JSON

Assim um objeto Autor com nome e e-mail ficaria assim em notação JSON:

{"nome":"Rodrigo","email":"rodrigocarreiro@rmfactory.com.br"}

Muito simples não é? E como dito anteriormente hoje em dia temos mais de 20 linguagens que suportam JSON. Agora onde o Delphi entra nisso? Como sempre facilitando a nossa vida. No Delphi 2010 temos uma nova classe chamada TJSONValue que, ao herdarmos dela, temos toda a infraestrutura necessária para que nosso objeto seja transportado via JSON. Ai surge outra questão: As classes que tenho hoje em meu framework terão também que herdar de TJSONValue?

A Embarcadero também pensou nisso e adicionou na VCL uma engine que converte nossas classes de negócio para a notação JSON e vice e versa. Vamos ver um exemplo rápido de como tudo funciona.

Crie uma nova aplicação no Delphi 2010 através do menu File->New->Other, na categoria DataSnap Server selecione a opção DataSnap Server. No diálogo que se abrir mantenha os valores padrão, marque a opção HTTP e clique em OK. Adicione uma nova unit ao projeto e nela crie uma nova classe chamada TAutor. Sua unit deverá estar parecida com o código da **Listagem 1**. Salve a unit como autor.pas.

Localize no projeto o ServerMethod que foi adicionado pelo Wizard, que nada mais é que um ServerModule, uma espécie de DataModule para DataSnap. Lá deverá conter um método de exemplo chamado EchoString. Vamos adicionar mais um que chamaremos de GetObjectToJSON. Na implementação deste Método coloque o código da **Listagem 2**. Não se esqueça de dar uses na unit autor.

Listagem 1. Declaração da classe Autor

```
unit autor;

interface

type
  TAutor = class
  private
    FEmail: String;
    FNome: String;
    procedure SetEmail(const Value: String);
    procedure SetNome(const Value: String);
  published
   property Nome: String read FNome write SetNome;
   property Email: String read FEmail write SetEmail;
  end;

implementation

procedure TAutor.SetEmail(const Value: String);
begin
  FEmail := Value;
end;

procedure TAutor.SetNome(const Value: String);
begin
  FNome := Value;
end;
end.
```

Listagem 2. Implemantação do método GetObjectToJSON

```
var
  Autor: TAutor;
  Converte: TJSONMarshal;
begin
  Autor := TAutor.Create;
  Autor.Nome  := 'Rodrigo Carreiro Mourão';
  Autor.Email := 'rodrigocarreiro@rmfactory.com.br';

  Converte :=  TJSONMarshal.Create(TJSONConverter.Create);
  Result := Converte.Marshal(Autor);
  FreeAndNil(Autor);
  FreeAndNil(Converte);
end;
```

Na **Listagem 2** nós criamos dois objetos, um do tipo TAutor que é o objeto autor que queremos transportar via JSON e o outro do tipo TJSONMarshal, que é a classe responsável pela conversão dos objetos nativos ou padrão para a notação JSON. A esse processo de conversão damos o nome de Marshalling. O método para converter um Object em um JSONObject chama-se Marshal e o processo inverso chama-se UnMarshal. Bem sugestivo.

Após declarar as variáveis criamos um objeto autor e damos alguns valores padrão para suas propriedades. Na sequência criamos um objeto TJSONMarshal passando como parâmetro no construtor uma instância de TJSONConverter. Por fim invocamos o método Marshal passando como parâmetro o objeto Autor. Retornamos o objeto pela função e destruímos as instâncias. Muito Simples.

Se você deixou as opções padrão nos componentes, então basta ativar os componentes DSServer e DSHTTPServer, alterando a propriedade Active deles para True. Execute a aplicação e dê permissão na tela do Firewall que irá se abrir. Para testar nossa aplicação vamos usar o contexto REST suportado pelo novo DataSnap. Agora podemos invocar os métodos de um servidor DataSnap via HTTP. Para isso temos que usar a seguinte sintaxe:

http://endereco_do_servidor/datasnap/rest/classe/metodo/parametro. Onde:

Figura 2. Objeto em notação JSON no browser

• endereço_do_servidor: Endereço IP ou Domínio onde o servidor está rodando;
• DataSnap: parâmetro padrão necessário para acessar o servidor DataSnap;
• rest: contexto rest no servidor;
• Classe: nome da classe que você quer acessar;
• Método: nome do método que você quer acessar na classe;
• Parâmetro: caso exista o parâmetro que deve ser passado para o método.

No nosso caso, como deixamos os nomes padrão sugeridos pelo IDE, nossa url ficará assim: http://localhost/datasnap/rest/TServerMethods1/getObjectToJson. Então com a aplicação rodando execute a URL acima no Browser de sua preferência. A **Figura 2** mostra o nosso objeto sendo exibido no browser em notação JSON.

Grande abraço e até a próxima! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Nesta sessão você encontra artigos intermediários sobre Delphi Win32 / .NET

SPED Contábil e Fiscal

Entenda como fazer a geração do arquivo solicitado pelo SPED

Do que trata o artigo
Este artigo visa ilustrar ao leitor, os procedimentos do SISTEMA PUBLICO DE ESCRITURAÇÃO DIGITAL (SPED), de tal forma, que, o leitor tenha uma introdução do funcionamento do mesmo, tornando-se apto a desenvolver soluções que atendam esta exigência.

Para que serve
O SPED serve para substituir os livros contábeis e fiscais, que são impressos pelas empresas, e também, para facilitar a consulta dos dados contábeis e fiscais das empresas, já que os mesmos terão armazenamento em meio digital.

Em que situação o tema é útil
Para empresas que precisam utilizar esta tecnologia imposta pelo governo federal, e para programadores que precisam desenvolver softwares de acordo com as exigências do manual de orientação para geração do arquivo.

Resumo do DevMan

O intuito deste artigo é mostrar as características do manual de desenvolvimento do arquivo, e com isso, levar o leitor a uma compreensão de como deverão ser constituídas as informações para o Governo. Construiremos um cadastro para exemplificar a geração do arquivo, iremos conhecer o software fornecido pelo governo federal para fazer validação e assinatura do arquivo gerado pelo ERP do usuário. Para isso, o governo brasileiro criou o Sistema Público de Escrituração Digital (SPED), que visa agilizar e eliminar a papelada das empresas no que diz respeito a diversos livros contábeis e fiscais além da nota fiscal eletrônica, e dessa forma, informatizar os aspectos burocráticos os quais todas empresas sofrem.

Jederson Zuchi
jederson@sygnux.com.br
Diretor de sistemas industriais da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP. Formado pela Fatec/TQ, desenvolve com Delphi desde a versão 3. Atualmente trabalhando com Delphi e com Java em diversos ERP's industriais.

Como apresentado, o SPED é um projeto criado pelo governo federal, que faz parte do Programa de Aceleração do Crescimento (PAC). Este projeto visa estreitar os laços entre o fisco e o contribuinte brasileiro. De modo geral, o SPED visa informatizar as obrigações impostas aos contribuintes, e modernizar o processo de entrega destes documentos para as secretarias

tributárias, gerando com isso facilidades nas consultas dos dados e também nas possíveis fiscalizações. Para garantir a validade jurídica desses documentos, mesmo estando em meio digital, se faz necessária a assinatura desses por meio de um certificado emitido por uma entidade certificadora, para que dessa forma, consigam garantir a integridade desses documentos perante o órgão competente.

Diante desses fatos, o SPED foi dividido em três grandes grupos, que são: Escrita Contábil Digital (ECD), a Escrita Fiscal Digital (EFD) e a Nota fiscal Eletrônica (NF-e). Escrita Contábil Digital – ECD - consiste basicamente na alteração dos livros contábeis, tais como o razão contábil e o diário geral, do formato impresso para o formato digital, ou seja, esses livros que fazem parte das obrigações acessórias das empresas, deixaram de existir em sua forma impressa e passaram a somente existirem na forma digital.

Em substituição a isso, o aplicativo do contribuinte deverá ser capaz de gerar um arquivo texto, respeitando as regras contidas no anexo único da Instrução Normativa RFB 787/07. O download dessa normativa está informado nos links ao final desse artigo, e ela poderá ser baixada para acompanhamento do layout na geração da ECD. Agora, iremos abordar alguns aspectos técnicos no que diz respeito à geração dessa normativa.

O arquivo gerado pelo software do contribuinte, deverá obrigatoriamente ser submetido ao Programa Validador e Assinador (PVA), disponibilizado para download pela Receita Federal, para ser validado e assinado da mesma forma como é o procedimento do SINTEGRA hoje. Portanto, nós, desenvolvedores, somente precisamos nos preocupar no resgate dos dados solicitados pelo layout no nosso software para gerar esse arquivo e, todo o restante do processo (assinatura, validação e transmissão) será feito pelo PVA. O segundo aspecto interessante é o formato do arquivo escolhido, que deve ser texto utilizando a codificação ASCII e ISO 8859-1. No arquivo texto não serão aceitos: campos compactados, binários e pontos flutuantes.

A estrutura de construção do arquivo deverá obedecer a um nível hierárquico, sempre iniciando um bloco e fechando-o, podendo, inclusive, haver um bloco dentro de outros blocos. Outra característica interessante é que os registros deverão iniciar com o caractere pipe "|" e terminar com o mesmo caractere, além de ser utilizado como caractere separador de registros. A **Figura 1** exibe um exemplo do arquivo da ECD, retirada da própria Instrução normativa RFB 787/07.

Perceba que no início de cada um dos blocos, os registros começam com o caractere "|", e que os blocos são iniciados com quatro caracteres que definem especificamente qual é o bloco que está sendo informado em uma determinada linha sendo depois precedido do caractere delimitador novamente. Observem também que quando uma informação não tem conteúdo, ela é precedida do caractere "||". Esta situação deverá acontecer para que a posição deste campo no layout não seja perdida. No exemplo que iremos construir a seguir, mostrarei uma situação desta para que seja observada e em ocorrências semelhantes, como a situação deve ser tratada.

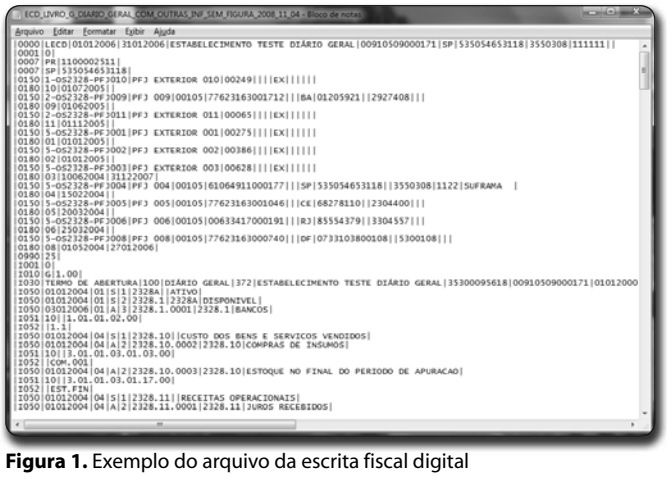


Figura 1. Exemplo do arquivo da escrita fiscal digital

O outro detalhe é que ao final de cada uma das linhas, o caractere CR/LF (Carrier Return/Line Feed) deve ser colocado para fazer o salto de linha e o retorno do "carro" para o início do texto. Estes caracteres correspondem respectivamente aos códigos 13 e 10 da tabela ASCII.

Outro ponto que deve ser destacado são os tipos de informações que nós podemos gerar para o arquivo da ECD. Eles podem ser de dois tipos: CARACTERES e NUMÉRICOS. Os campos caracteres podem ser todos os caracteres da tabela ASCII, com exceção dos caracteres não imprimíveis (a exemplo do ENTER, das teclas F1, F2, F3..., ESC etc) e também do caractere "|" que é utilizado para fazer a separação entre um registro e outro dentro do layout. Quanto aos numéricos, devem ser todos os números de 0 a 9, incluindo o campo de vírgula para fazer a separação da parte inteira da parte fracionária de cada campo. Quando falamos em casas decimais, deve-se observar o limite para elas, de acordo com cada campo. Considera-se também os dados com o tipo DATA como caracteres numéricos, já que a data de 01/01/2010 é representada como sendo 01012010. Caso a data seja somente, por exemplo, mês/ano, continuamos tratando da mesma forma, retirando as barras e gerando somente os números, ou seja, se no caso tivemos a seguinte situação 12/2009, é gerado no arquivo somente 122009. Como vimos, retira-se as barras e formata-se o campo como sendo um numero inteiro.

Criando um cadastro de exemplo para gerar o arquivo

Agora que estamos cientes das normas técnicas da criação do arquivo, vamos fazer um cadastro básico para poder criar uma parte do arquivo, que servirá de exemplo para podermos criar todos os registros. O cadastro escolhido para ser criado é o cadastro de plano de contas contábeis, que é representado no arquivo pelo bloco I050. Cadastraremos algumas contas em uma tabela temporária, utilizaremos para isso o ClientDataSet que está disponível na Tool Palette do Delphi 2010.

Iniciaremos agora o nosso projeto de exemplo. Com o Delphi 2010 aberto, criaremos um projeto do tipo VCL Forms Application – Delphi. Com o formulário do novo projeto já visualizado, colocaremos os componentes necessários para fazermos o cadastro do plano de contas. Os componentes que

iremos utilizar são um ClientDataSet, um DataSource, um DBNavigator e por fim um DBGrid. A tela de cadastro deve ficar conforme a **Figura 2**.

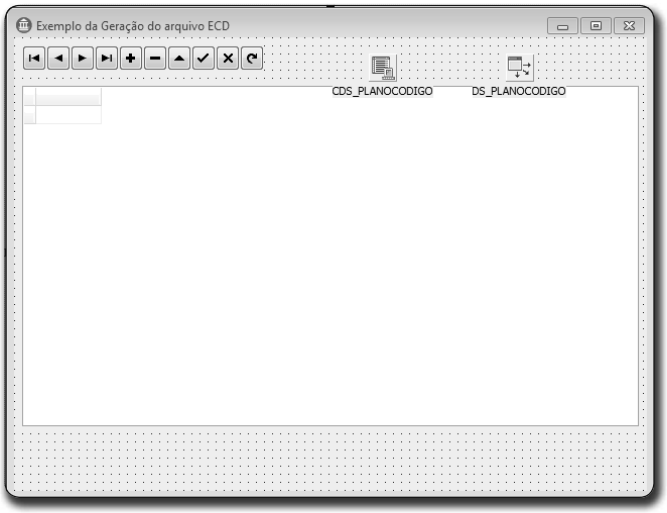


Figura 2. Formulário básico de cadastro do plano de contas com os componentes

Percebam que ao final do formulário deixei um espaço em branco de forma proposital para que implementemos ali, funções da geração do arquivo da ECD. No entanto, vamos agora nos ater somente ao cadastro do plano de contas.

Com os componentes devidamente inseridos, vamos agora começar a configurar nosso sistema. Vamos criar os campos temporários que irão compor a estrutura da nossa tabela temporária no ClientDataSet, para isso, dê dois cliques no componente CDS_PLANOCODIGO e, após isso, na janela que se abrirá, escolha a opção NEW FIELD. Se os passos foram executados corretamente, deverá ser aberto um formulário, igual ao representado na **Figura 3**.

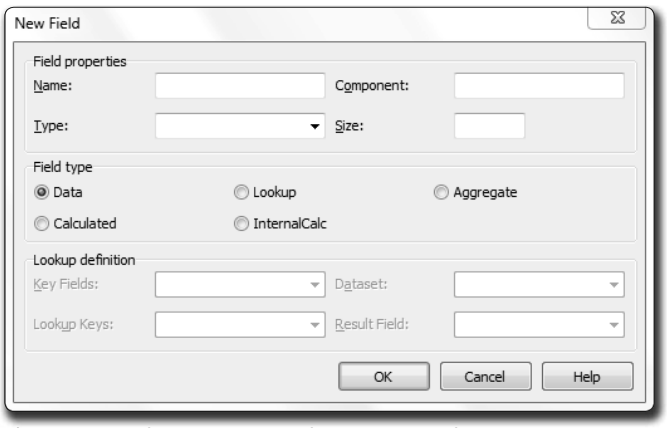


Figura 3. Formulário para criação dos campos no ClientDataSet

Após preencher os campos no formulário, clicamos em OK. É importante lembrar que os campos que devem ser preenchidos são: Name, Type, e Size (de acordo com o tipo que escolhemos), e no grupo do Field Type, devemos necessariamente escolher a opção Data. Na **Tabela 1** estão relacionados todos os campos que precisaremos criar para que possamos prosseguir com o nosso cadastro.

Número	Nome (Name)	Tipo (Type)	Tamanho (Size)	Descrição
1	DtMovto	Date	----	Data da Alteração/ Inclusão
2	CodNatureza	String	2	
3	TipoConta	String	1	
4	Nivel	Integer	----	
5	CodigoCta	String	35	
6	CodigoCtaAnt	String	35	
7	Descricao	String	50	

Tabela 1. Campos do cadastro

Como mostrado na tabela anterior, precisaremos agora criar e colocar esses campos no componente ClientDataSet, para que possamos gerar o cadastro de acordo com o que é solicitado no layout da instrução normativa da ECD. Após todos os campos criados, iremos agora criar a tabela temporária, que é um recurso que o ClientDataSet nos fornece. Para isso, basta clicar sobre o mesmo e com o botão direito do mouse selecionar a opção CreateDataSet. Após esse procedimento, o ClientDataSet estará pronto para receber as informações e armazená-las em memória.

O nosso próximo passo é fazer a ligação dos componentes DBGrid e DBNavigator ao DataSource, para que consigamos fazer a manipulação dos dados do nosso exemplo.

Após serem feitas as ligações dos componentes, vamos inserir as colunas da DBGrid, para que possamos utilizar alguns recursos dela. Para inserir as colunas na DBGrid, dê um duplo clique sobre a mesma e adicione sete colunas, que serão ligadas aos campos do nosso ClientDataSet. No campo CodNatureza, clique na propriedade PickList e insira as informações, conforme podemos visualizar na **Figura 4**.

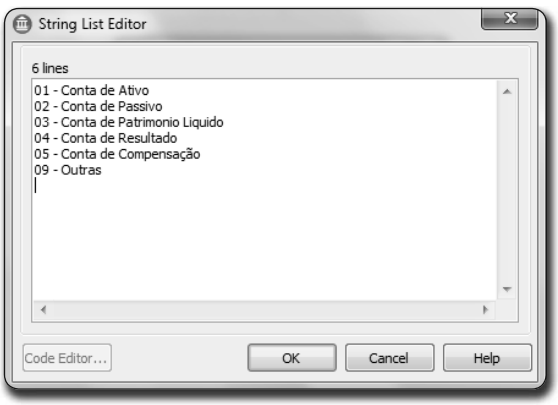


Figura 4. Propriedade PickList do campo CodNatureza

As informações que estão na **Figura 4** foram retiradas do menu ajuda do PVA, que é outra fonte interessante de informações, principalmente quando precisamos consultar as tabelas que os blocos de registros nos solicitam.

O recurso do PickList também deve ser utilizado no campo tipo de conta, que deverá conter somente duas linhas com as seguintes informações: Na primeira linha S – Sintético e na segunda linha A – Analítico. Os demais campos podem ser incluídos, mas sem nenhuma observação especial.

Com isso nosso cadastro está pronto para ser utilizado. O próximo passo é criar algumas funções de validação dos dados, para que possamos gerar as informações de acordo com que é solicitado na instrução normativa. Essas funções serão a função de validação de dados numéricos e alfanuméricos, como mostra a **Listagem 1**.

Listagem 1. Função para validar os campos numéricos

```
function TForm1.FormataCampoNumeric(Campo: String): String;
var
  i : Integer ;
begin
  Campo := StringReplace(Campo, '/','',[rfReplaceAll,rfIgnoreCase]);
  Campo := StringReplace(Campo, '.','',[rfReplaceAll, rfIgnoreCAse]);
  i := 1 ;
  while (i <= Length(Campo)) do
  begin
    if not (Ord(Campo[i]) in [44, 48..57] )then
    begin
      ShowMessage('Campo numérico com caractere inválido ' +
        QuotedStr(Campo[i]));
      Abort ;
    end;
    inc(i);
  end;
  Result := Campo;
end;
```

A função correspondente a **Listagem 1** realiza a formatação dos campos de data para numéricos e também retira os pontos, caso o número que está entrando seja do tipo Currency. Em seguida, ela fará a validação para verificar se todos os caracteres estão dentro do padrão, que são os caracteres vírgula, código 44 da tabela ASCII e, também, os números que na tabela ASCII correspondem às posições do número 48 a 57. Caso os campos sejam inválidos, no nosso caso estamos jogando uma mensagem na tela, mas vocês poderão fazer outros tratamentos, como, por exemplo, retirar essa informação ou gerar um arquivo com as inconsistências, fica a ideia. No entanto, se não forem encontrados caracteres inválidos, é retormado o Result da variável campo para que possamos utilizar essa informação validada para fora da função.

A próxima função diz respeito aos caracteres alfanuméricos que de acordo com a normativa, os campos que devem ser verificados são os caracteres "|" (código 124 na tabela ASCII) e os campos não imprimíveis (códigos da tabela ASCII de 00 a 31), todos os outros caracteres são permitidos. Na **Listagem 2** é mostrada a função.

Assim como na outra função da **Listagem 1**, caso haja algum caractere inválido, somente mostramos este na tela, mas poderíamos também tratá-los de alguma outra forma. Fica a seu critério decidir o que fazer.

Agora mostrarei outra função que diz respeito à formatação da linha que deverá ser gravada no arquivo da ECD. Esta terá como principal característica a colocação dos caracteres pipes entre os campos para que fiquem de acordo com o solicitado nos requisitos do arquivo. Na **Listagem 3** demonstro esta função que nos facilitará quando formos criar os arquivos.

Listagem 2. Código referente à formatação de campos alfanuméricos

```
function TForm1.FormataCampoString(campo: String): String;
var
  i : Integer ;
begin
  i := 1 ;
  while (i <= Length(Campo)) do
  begin
    if (Ord(Campo[i]) in [00..31, 124] )then
    begin
      ShowMessage('Campo alfanumérico com conteudo inválido: ' +
        IntToStr(Ord(Campo[i])) );
       Abort ;
    end;
    inc(i);
  end;
  Result := Campo ;
end;
```

Listagem 3. Função para formatação dos campos entre os pipes

```
function TForm1.GeraStringFormatoSped(Campo: String;
  Registro: String): String;
begin
  if Length(Registro) > 0  then
  begin
    if Registro[Length(Registro)] <> '|' then
      Registro := Registro + '|';
  end
  else
    Registro := '|' ;
  Registro := Registro + Campo + '|';
  Result := Registro;
end;
```

A função da **Listagem 3**, tem por objetivo colocar os registros, entre os pipes, conforme solicita a instrução normativa para gerar a ECD.

Após todos esses procedimentos seguidos, vamos partir para a geração do arquivo da escrita contábil digital. Iremos agora, criar a rotina responsável por gerar os registros para exportarmos no formato da ECD. Para complementar nosso cadastro, colocarei mais dois componentes, que servirão para nos auxiliar na geração do arquivo. Esses componentes serão um SaveDialog e um SpeedButton, para que possamos gerar e salvar o arquivo em um lugar desejado pelo usuário. No SpeedButton, codifique o evento OnClick conforme o o código da **Listagem 4,** para que ele possa gerar as informações no layout adequado.

Com a codificação para geração do arquivo já realizada, podemos executar o programa e cadastrar alguns itens para fazermos os testes. Na **Figura 5**, apresento alguns dados de testes para que possamos gerar o arquivo de acordo com o layout solicitado.

Na **Figura 5** observamos um cadastro básico de um plano de conta para podermos gerar o arquivo da escrita contábil digital, lembrando que esse teste, somente irá gerar o bloco I050 que é o bloco de plano de contas, portanto, não será um bloco validável no PVA. Agora para gerarmos o arquivo, basta clicarmos no botão "Gerar Arquivo", e após este procedimento o arquivo será gerado ficando de acordo com a **Figura 6**.

Como podem observar, nos campos do primeiro nível não existe conta anterior e devido a isso, a posição pertinente está sendo gerada com conteúdo vazio. Como já vimos, isso precisa ser feito para que os campos não percam as respectivas posições do layout de um determinado bloco.

Listagem 4. Gerando o arquivo da ECD

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
var
  Registro : String ;
  ArqText : TextFile ;
begin
  if CDS_PLANOCODIGO.RecordCount = 0  then
  begin
     MessageDlg('Não existem registros de plano de conta cadastrado',
                  MtInformation, [MBOK],0);
     Abort ;
  end;
  if not SaveDialog1.Execute then
    Abort ;
  AssignFile(ArqText, SaveDialog1.FileName);
  Rewrite(ArqText) ;  CDS_PLANOCODIGO.First ;
  while not CDS_PLANOCODIGO.Eof do
  begin
    Registro := '';
    Registro := GeraStringFormatoSped( 'I050', Registro );
    Registro := GeraStringFormatoSped( FormataCampoNumeric(
                FormatDateTime('DD/MM/YYYY',
               CDS_PLANOCODIGO.FieldByName('DtMovto').AsDateTime)),
                registro );
    Registro := GeraStringFormatoSped(FormataCampoString(
               CDS_PLANOCODIGO.FieldByName('codNatureza').AsString ),
                registro ) ;
    Registro := GeraStringFormatoSped( FormataCampoString(
               CDS_PLANOCODIGO.FieldByName('tipoConta').AsString),
                registro ) ;
    Registro := GeraStringFormatoSped(FormataCampoNumeric(
                CDS_PLANOCODIGO.FieldByName('nivel').AsString),
                registro);
    Registro := GeraStringFormatoSped(FormataCampoString(
                CDS_PLANOCODIGO.FieldByName('CodigoCta').AsString),
                registro ) ;
    Registro := GeraStringFormatoSped(FormataCampoString(
               cds_planoCodigo.FieldByName('codigoCtaAnt').asString),
                registro );
    Registro := GeraStringFormatoSped(FormataCampoString(
                CDS_PLANOCODIGO.FieldByName('Descricao').asString),
                registro);
    Writeln(ArqText, registro) ;
    CDS_PLANOCODIGO.Next ;
  end;
  CloseFile(ArqText);
  MessageDlg('Arquivo finalizado e concluido com sucesso',
  MtInformation, [MbOK],0);
end;
```

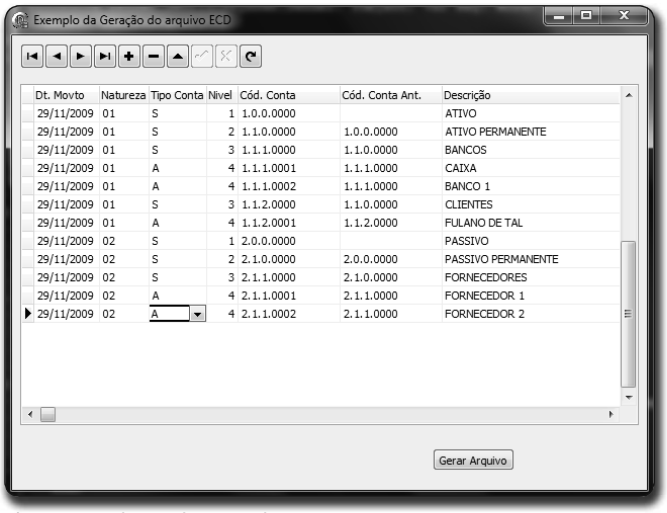


Figura 5. Cadastro de exemplo para a ECD

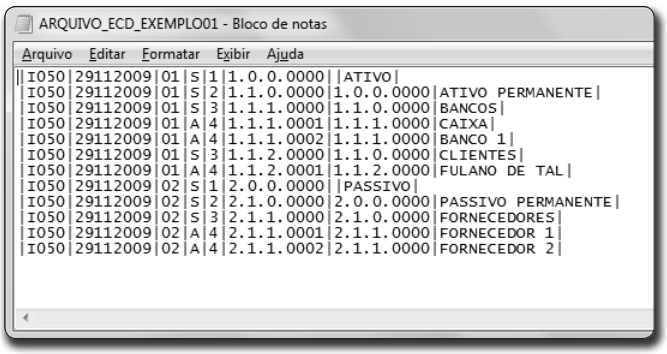


Figura 6. Estrutura do arquivo ECD gerado de acordo com nosso exemplo

Programa Validador Assinador

Com o arquivo gerado, o próximo passo é passá-lo pelo Programa Validador e Assinador (PVA), que está disponível no site do SPED na Receita Federal do Brasil (verifique na sessão Links ao final do artigo). Conforme já informei, o arquivo gerado pelo nosso sistema deve sistematicamente passar pelo PVA para que o mesmo possa ter sua estrutura submetida à validação, e também a assinatura. Segundo a instrução normativa, é obrigatório o uso do PVA para esses processos, além do processo de transmissão do arquivo para o órgão competente.

Nota

Este software está disponível para ambiente Windows e também para ambiente Linux, e roda sobre a Virtual Machine do Java. Para rodar o PVA é necessário ter a JVM (Java Virtual Machine) instalada.

Após ser feita a instalação, o PVA já está pronto para ser utilizado. Abrindo-se o software (que é todo em ambiente gráfico), o usuário irá se deparar com um menu superior com as opções desejadas. O primeiro processo é fazer a validação do arquivo gerado pelo software. Nessa validação o PVA irá retornar todos os possíveis problemas que o arquivo da ECD apresenta, para que nós possamos efetuar as correções. Depois de validado, o próximo passo é fazer a assinatura digital desse arquivo, e para isso, devemos ter instalado na máquina um certificado digital para que possamos garantir a autenticidade desse arquivo digital. Na **Figura 7** é mostrado um relatório de um arquivo já validado pelo PVA.

Podemos observar na **Figura 7** o relatório gerado após a validação do arquivo, que neste caso não encontrou nenhum erro estrutural, nem de conteúdo nos registros do arquivo da ECD.

O próximo passo é a execução da assinatura, porém, não possuo um certificado de teste para a assinatura do arquivo pelo PVA. Assim que assinado, este arquivo está apto a ser transmitido para a Receita Federal.

Outra função interessante no PVA, diz respeito ao seu menu de ajuda. Ali podemos encontrar diversas tabelas que o layout da ECD nos solicita, sendo uma rica fonte de informação para o desenvolvedor, além de conter também a instrução normativa no seu menu de ajuda. Dessa forma, quem precisar desenvolver algo com relação à ECD, deve

ter o PVA instalado também para consulta de dados e não só para validação.

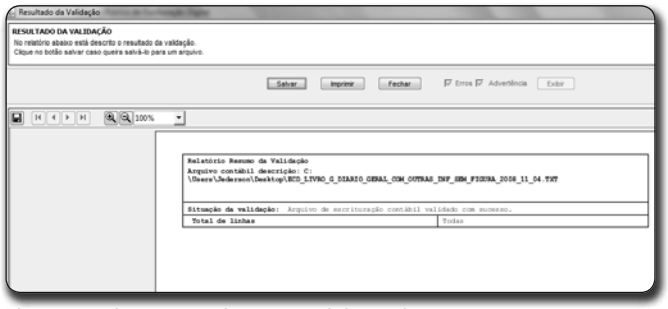


Figura 7. Relatório gerado após a validação do arquivo ECD

Escrita Fiscal Digital - EFD

A escrita fiscal digital (EFD) é o outro braço do SPED, que tem por objetivo, reunir em um arquivo as informações das escriturações fiscais, bem como informações de interesse do fisco, como, por exemplo, os registros de apurações dos impostos gerados pelo contribuinte.

Da mesma forma que a ECD, a escrita fiscal consiste em retirarmos informações de nossos softwares e gerar um arquivo de mesmo aspecto técnico para ser submetido ao programa validador e assinador, porém, apesar do programa ter o mesmo nome, ambos são softwares distintos, pois um diz respeito à contabilidade e outro à parte fiscal.

O layout da EFD está definido em ato cotepe e este pode ser baixado no site da EFD, que também se encontra na sessão de Links ao final do artigo. O ato cotepe visa orientar e mostrar o layout para gerar o arquivo da escrita fiscal digital.

Há uma grande quantidade de informações que devemos gerar para a EFD, e também diversas tabelas externas que devemos associar a nosso software, já que elas são utilizadas para gerar o arquivo digital EFD.

Um detalhe interessante - e motivo de muitos erros por parte de nós programadores, na qual refere-se à ordenação no momento da geração do arquivo da EFD - é que essas informações devem ser geradas em ordem cronológica crescente dos acontecimentos, ou seja, devem começar da mais antiga para a mais nova. No manual da EFD, devemos observar a ocorrência que cada bloco poderá apresentar, e em alguns casos, poderemos ter blocos que podem ser informados uma única vez no arquivo.

Outro ponto que a EFD se diferencia da ECD, é que cada estado poderá solicitar determinadas informações, já que, um estado se diferencia do outro no que diz respeito à obrigações acessórias e dessa forma, devemos tomar muito cuidado sempre verificando a legislação vigente para cada estado.

Quanto a geração deste arquivo, a sua periodicidade de apresentação junto à Receita Federal é mensal, portanto, todo mês o sistema deve ser capaz de gerar essas informações. Quem está obrigado a entregar o Sintegra já faz uma geração parecida, e agora, se estivermos na lista de empresas obrigadas a gerar as informações da EFD, também deveremos apresentá-las mensalmente. Para sabermos quanto à obrigatoriedade da entrega ou não, devemos consultar as secretarias das fazendas estaduais, para fazermos a consulta. Se não estiver disponível a consulta nas secretarias, podemos consultar no próprio site da Receita Federal, já que existe uma lista completa de contribuintes obrigados à entrega da EFD.

Os blocos do arquivo da EFD iniciam-se em 0000 e terminam em 9999, porém podemos identificar os blocos por sua natureza (primeiro caractere do conjunto de quatro dígitos do bloco). A **Tabela 2** mostra a natureza de cada um dos blocos.

Natureza	Descrição
0	Abertura, Referência e Identificação
C	Documentos Fiscais I – Mercadoria ICMS e IPI
D	Documentos Fiscais II – Serviços ICMS
E	Apuração do ICMS e do IPI
H	Inventário Físico
I	Outras informações
9	Controle e encerramento do arquivo

Tabela 2. Natureza dos blocos

Isso é importante para verificarmos também a sequência que os blocos devem ser gerados, já que devem iniciar pelo bloco 0 e terminar no bloco 9, sempre seguindo essa ordem. Para desenvolvermos também o arquivo da EFD, podemos utilizar as funções apresentadas nas **Listagens 1**, **2** e **3** já que a parte técnica da construção do arquivo é a mesma.

Após feita a elaboração do arquivo e este for submetido à validação do PVA, somente poderão assinar estes arquivos os certificados e-PJ ou e-CNPJ e para o representante legal, o e-PF ou o e-CPF, ambos os certificados emitidos por entidade competente de acordo com a ICP-Brasil.

Com isso verificamos que os aspectos técnicos para a geração do arquivo da EFD não são trabalhosos e todo o processo de validação, assinatura e transmissão devem ser feitos pelo PVA que é disponibilizado no site da Receita Federal. Diferentemente da nota fiscal eletrônica, que é um processo mais automatizado e de mais fácil integração com o ERP de cada um, os arquivos da EFD são processos mais manuais, com exceção da geração do mesmo.

Os sistemas devem estar preparados e com todas as tabelas informativas ao dispor, já que elas, como já dito, são pré-requisitos para a geração do arquivo.

Projeto ACBR

Há um projeto que está na internet, chamado ACBR. Inclusive é o mesmo projeto que desenvolveu os

componentes do ACBR-NFe, que foi apresentado aqui na revista há algumas edições. Este projeto está desenvolvendo também todas as classes de informações para a geração dos arquivos do SPED.

Ainda não é algo que podemos considerar em produção, mas o pessoal que desenvolve este componente, que por sinal é Open-Source, provou várias vezes a sua competência, e hoje temos várias empresas já rodando a solução proposta pelo componente no que diz respeito à nota fiscal eletrônica.

Para quem tem o repositório de dados do projeto, é possível verificar que o mesmo está em desenvolvimento constante e é notável que todos os dias surjam atualizações no que tange o SPED Contábil e Fiscal.

Desta forma podemos esperar algumas classes já automatizadas para a geração, validação (não oficial) dos dados para o SPED diretamente do nosso sistema. Para quem quer acompanhar este importantíssimo projeto, basta acessar o fórum dos projetos ACBr, e verificar as dúvidas, os problemas que estão surgindo no desenvolvimento, e também, até contribuir com os membros do projeto no que diz respeito à construção das classes.

Conclusão

Com as exigências que o governo federal vem nos trazendo, teremos que, cada vez mais lançarmos sistemas automatizados, e que consigam nos gerar informações em tempo real, para conseguirmos atender as exigências. Outro ponto que podemos destacar, é que aos olhos contábeis e fiscais a tendência é que cada vez mais os softwares se tornem padronizados, pelo menos nessa área, já que irão compartilhar de diversas informações em comum, como por exemplo, tabelas de planos de contas, países, municípios e demais informações. Agora mais do que nunca temos que nos ater a parte legislativa, para conseguirmos ter as informações corretas, mas da maneira com que o governo possa visualizar. ●

Links

Site do SPED na Receita Federal do Brasil
www.receita.fazenda.gov.br/Sped

Link da Instrução Normativa da RFB 787/07 para Download
http://www.receita.fazenda.gov.br/publico/Legislacao/Ins/2007/AnexoUnicoINRFB787.doc

Download do PVA Contábil e Fiscal
http://www.receita.fazenda.gov.br/Sped/Download/Default.htm

Lista dos obrigados a entregarem a EFD
http://www.fazenda.gov.br/confaz/confaz/Diversos/Lista_Atualizada_Jan2009_Obrigados_EFD_2009.pdf

Fórum do Projeto ACBR - SPED Contábil
http://www.forumweb.com.br/foruns/index.php?/topic/79211-iniciar-sped-contabil/

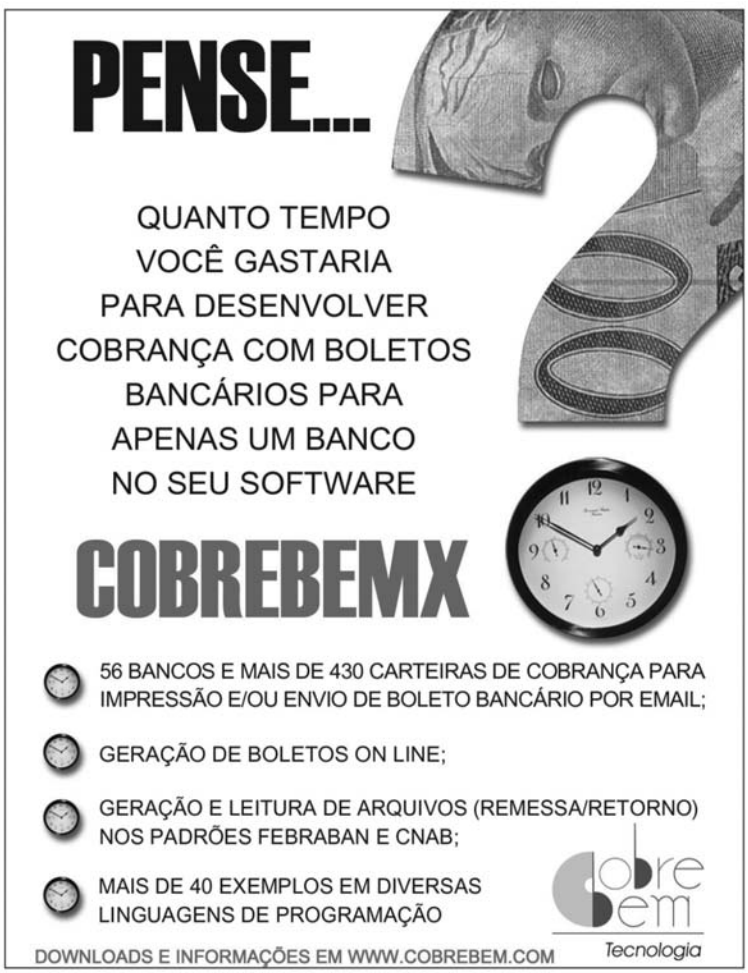
Fórum do Projeto ACBR - SPED Fiscal
http://www.forumweb.com.br/foruns/index.php?/topic/78912-sped-fiscal/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

POO

Utilize Generics, Padrões de Projeto e OO para criar um Log das ações do sistema

Do que trata o artigo

Este artigo mostra como criar um algoritmo para gerenciar os logs do sistema usando as técnicas orientadas a objetos. Além disto, durante todo o artigo é discutido sobre qual a melhor forma de se criar um sistema flexível.

Para que serve

Ter uma boa definição da responsabilidade de cada parte do seu sistema tornará o seu trabalho um pouco mais 'suave'. Sistemas com códigos bem organizados geralmente facilitam o trabalho da equipe de desenvolvimento e permitem que seja dispensado mais tempo em outras tarefas secundárias, como por exemplo, um log das ações executadas (apresentado neste artigo).

Em que situação o tema é útil

Log de ações é um recurso que pode ajudar você a entender melhor seu próprio sistema prevenindo comportamentos estranhos antes mesmo de o cliente descobri-los. Junte isto com as vantagens de se ter um sistema bem organizado e o resultado será um programa de qualidade no mínimo considerável pelos concorrentes.

Resumo do DevMan

Organização! Esta é a palavra de ordem para um programa com qualidade. Não estou falando da organização visual do programa, pois esta é intrínseca. Estou falando da organização estrutural dos programas: a organização do seu código vai definir a qualidade do seu programa. Um código mal feito gera consequências ruins para o programa e em alguns casos quase instantaneamente. Normalmente a primeira consequência de um código desorganizado é a inflexibilidade do sistema: de fato o sistema se torna mais rígido e as alterações futuras ficarão cada vez mais difíceis. Este artigo explica como criar códigos flexíveis usando como exemplo uma aplicação que precisa ter um histórico das ações executadas.

Rafael Stavarengo
faelsta@gmail.com
Programador-analista da Cheina Informática. Trabalha com Delphi, JME e PHP + ZendFramework. Formando em Técnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.
http://www.devmedia.com.br/stavarengo

Você já teve a sensação de que existe uma maneira mais elegante de codificar um algoritmo que você acabou de criar? Eu também já tive esta impressão e geralmente existe mesmo uma maneira melhor. Não adianta criar um algoritmo, seja ele qual for, se ele não estiver bem estruturado e organizado. Por mais que um algoritmo

pareça ser ótimo ele irá se desgastar com o tempo, conforme as alterações forem acontecendo.

Existem vários padrões e princípios de programação orientada a objetos que tentam diminuir este desgaste natural dos algoritmos. Neste artigo eu vou mostrar como pode ser feito um algoritmo para log das ações do sistema, que seja flexível o suficiente para atender as necessidades de mudanças que o dia-a-dia impõe.

Aprenda a identificar códigos que devem ser separados

Vamos direto ao ponto: crie um novo projeto no Delphi 2010 (File>New>VCL Forms Applications). No formulário principal que irá abrir adicione quatro TButton e um TListBox. Configure as propriedades destes componentes de acordo com a **Tabela 1**. Quando você terminar de configurar os componentes do formulário, tente deixá-los como está apresentado na **Figura 1**.

Componente	Propriedade	Valor
TButton	Name	btAcao1
	Caption	Ação 1
TButton	Name	btAcao2
	Caption	Ação 2
TButton	Name	btAcao3
	Caption	Ação 3
TButton	Name	btExibirLog
	Caption	Exibir log
TListBox	Name	lbLogDoSistema

Tabela 1. Componentes

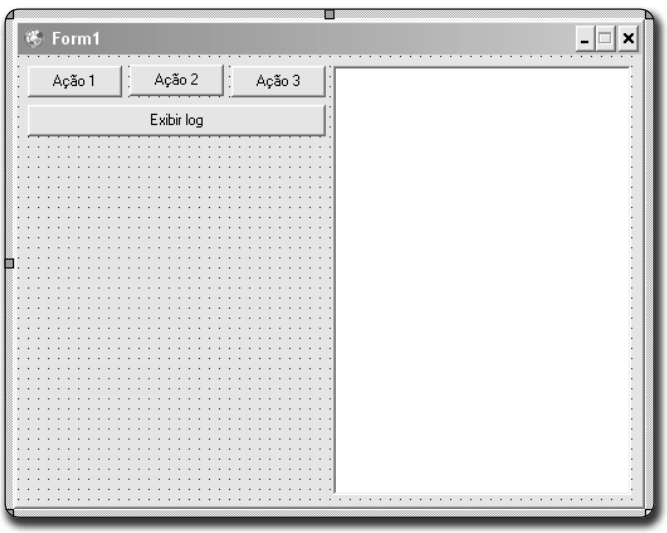


Figura 1. Primeiro layout do formulário principal

Os botões btAcao1, btAcao2 e btAcao3 serão os responsáveis por gerar um log no sistema. Eles representam as ações que programas do mundo real executam como, por exemplo, salvar ou excluir um registro. Também é comum gerar um log de tudo que acontece de errado no sistema: estes botões de ações representam qualquer ação que um programa de verdade poderia criar logs. O botão btExibirLog deve exibir uma lista de todos os logs do sistema no TListBox chamado lbLogDoSistema.

Neste caso é comum que o programador crie um método chamado SalvarLog dentro deste formulário e para cada clique no botão seria feito uma chamada ao método SalvarLog. Esta pode ser uma prática comum, porém, muito desorganizada, pois isto cria pelo menos uma consequência ruim instantaneamente. Se você precisar criar um log a partir de outros lugares do sistema (e você vai precisar), você terá duas escolhas:

a) Copiar e colar o método SalvarLog no lugar onde você está precisando dele: isto irá criar um caos na manutenção do log do seu sistema, pois qualquer alteração nesta parte obrigará você a procurar por todas as partes do sistema onde o log é manipulado para fazer a alteração;

b) Criar uma referência ao formulário onde o método SalvarLog foi criado: a princípio esta opção parece menos agressiva, mas não é. Esta decisão viola mais de um princípio da orientação a objetos. Neste caso quero destacar a violação do princípio que nos diz que uma classe só deve ter uma única razão para mudar. Mas, na verdade estaremos dando duas razões para a classe TForm1 mudar. A primeira seria alguma mudança no código que só diz respeito a ela mesma, como por exemplo, adicionar uma pergunta antes de fechar a janela – até aqui tudo certo. A segunda razão seria alguma mudança na maneira como log é gerado e você deve concordar que pelo fato da classe TForm1 já ter outras coisas para se preocupar não é justo que ela também se preocupe com a maneira como o log é gerado.

Nota

Conheça o princípio de responsabilidade única. Ele diz que toda classe deve ter apenas uma responsabilidade. Em outras palavras, toda classe deve ter apenas uma única razão para sofrer alterações.

A solução para este impasse é a separação destes códigos. Mas depois de separado para onde vai? Neste exemplo nós temos apenas uma classe (até agora). Em sistemas de verdade é comum existir dezenas de classes e até mesmo units criadas sem classe alguma. Em um sistema como este, onde todas as classes já têm sua responsabilidade (às vezes mais de uma e isto está errado), para onde deveria ir este novo código que manipula os logs do sistema? Talvez para uma destas units que possuem apenas métodos sem classes declaradas? Não. Esqueça esta ideia de criar units apenas para armazenar métodos. Esta é uma técnica que a própria equipe de desenvolvimento do Delphi está evitando – é possível perceber isso a cada nova atualização. Estamos falando de programação orientada a objetos, então TODAS as units devem ter uma classe declarada e TODOS os métodos também devem pertencer a uma classe específica. Por esta razão, o código separado deve ir para uma nova classe: então será criada uma nova classe para onde irá o código responsável por manipular os logs.

Separando a parte do código que muda

Como foi dito no tópico anterior a classe TForm1 não deve se responsabilizar pelos códigos de manipulação dos logs, por isto, irei criar uma nova classe somente para cuidar deles (os logs).

Dentro de uma nova unit vou criar uma classe chamada TLogDao: esta classe será a responsável por gerenciar tudo que se diz a respeito de logs. Esta será a responsabilidade dela. A única responsabilidade dela. Através da classe TLogDao qualquer outra classe do sistema poderá salvar uma mensagem de log ou buscar as mensagens que já foram salvas. Veja **Listagem 1**.

Listagem 1. A classe TLogDao

```
1 unit Log;
2 Interface
3 uses Generics.Collections;
4 Type
5   TLog = record
6     DataHora: TDateTime;
7     Mensagem: string[255];
8   end;
9   TLogFile = file of TLog;
10   TLogDao = class
11   Private
12     function ArquivoDeLog: string;
13   Public
14     procedure Salvar(Log: TLog);
15     function Buscar: TList<TLog>;
16   end;
17 Implementation
18 {Implementação dos métodos}
```

Nota

Se você está criando este projeto de exemplo junto comigo, fique calmo: a implementação dos métodos declarados na **Listagem 1** está no final do artigo.

Nota

Normalmente o tipo TLog (linha 5) seria na verdade uma classe, porém, para evitar códigos muito extensos eu preferi usar records no lugar de classes. Mas isto não muda em nada o objetivo do artigo. Veja logo abaixo, na linha 10 da **Listagem 1**, que a classe chamada TLogDao é declarada como havia dito.

A **Listagem 1** mostra o cabeçalho da classe TLogDao. Observe que existem apenas dois métodos públicos nesta classe, são eles: Salvar na linha 14 e Buscar na linha 15. Eles fazem o que o próprio nome sugere. O método Salvar recebe uma variável do tipo TLog declarado entre as linhas 5 e 8 e salva as informações desta variável, não se sabe como ou onde. Como e onde não é importante, pois isto é a responsabilidade da classe TLogDao. O que você precisa saber é quais são os métodos públicos desta classe e qual o objetivo deles. Falando nisto, o método Buscar por sua vez, é o responsável por buscar as informações que o método Salvar gravou. Repare que mais uma vez não é importante saber como ou onde estas informações estão, pois isto não é responsabilidade sua, mas sim do programador que criou a classe TLogDao. O importante sobre o método Buscar é que todas as mensagens de logs que já foram salvas serão retornadas em uma lista.

Agora que você entendeu qual a responsabilidade e como funciona a classe TLogDao, chegou a hora de usá-la no formulário principal, que também é conhecido como classe TForm1. Lembre-se de que havia um problema nesta classe, pois além de ser responsável por controlar as suas próprias funções também havia a necessidade de controlar as funções relacionadas ao gerenciamento dos logs. Você é capaz de me dizer qual é a verdadeira responsabilidade da classe TForm1 apenas olhando para a **Figura 1**? Eis a resposta: a classe TForm1 é responsável por fazer o formulário Form1 funcionar. Apenas isto! Veja na **Listagem 2** como fica a classe TForm1 após a criação da classe TLogDao.

Nota

Generics é um dos recursos que eu mais gosto de usar. É uma técnica fantástica que facilita seu trabalho de tal forma que após escrever as suas primeiras linhas de código usando generics você se pega pensando: "como pude viver tanto tempo sem isto?". Para saber mais sobre generics, veja os links no final do artigo.

Listagem 2. A única responsabilidade da classe TForm1

```
1   Implementation
2   uses Log, Generics.Collections;
3   {$R *.dfm}
4   procedure TForm1.btAcao1Click(Sender: TObject);
5  Begin
6    CriarLog('Ação 1');
7  end;
8  procedure TForm1.btAcao2Click(Sender: TObject);
9   Begin
10    CriarLog('Ação 2');
11  end;
12  procedure TForm1.btAcao3Click(Sender: TObject);
13  Begin
14    CriarLog('Ação 3');
15  end;
16  procedure TForm1.CriarLog(Mensagem: string);
17  Var
18    Dao: TLogDao;
19    Log: TLog;
20  Begin
21    Log.Mensagem := Mensagem;
22    Dao := TLogDao.Create;
23    Dao.Salvar(Log);
24    FreeAndNil(Dao);
25  end;
26  end.
```

Nota

Se você está criando este exemplo junto com o artigo, não se esqueça de declarar as units na sessão uses como pode ser visto na linha 2. Lembre-se: este exemplo foi feito em Delphi 2010. O código da **Listagem 2** não compila em versões do Delphi que não tenham suporte ao generics.

Ficou simples, você concorda? Tente imaginar como seria se TForm1 também fosse responsável por salvar e buscar as informações do log? Pense, como seria se os métodos declarados nas linhas 14 e 15 da listagem 2 também fizessem parte da **Listagem 2**.

Vamos aos fatos. A **Listagem 2** mostra a implementação da classe TForm1 cuja responsabilidade é executar os métodos OnClick de cada botão presente no formulário. Entre as linhas 4 e 15 temos as chamadas dos botões que representam as ações que um sistema normal poderia ter (veja segundo parágrafo do tópico 'Aprenda a identificar códigos que devem ser separados'). Em sistemas reais estes botões estariam fazendo coisas mais complexas, porém para este exemplo eles apenas criam um registro de log. Para evitar a duplicação de código que existira entre os três botões de ação, eu criei um método privado chamando CriarLog na classe TForm1 cujo objetivo é gerar um log. Você pode ver a implementação deste método entre as linhas 16 e 25.

Olhe para a **Figura 1** novamente: se a classe TForm1 é responsável por executar os métodos OnClick de cada botão, então no código apresentado na **Listagem 2** está faltando a implementação do método OnClick do botão lbLogDoSistema. Veja este código na **Listagem 3**

Listagem 3 Exibindo as mensagens dos logs existentes

```
1 procedure TForm1.btExibirLogClick(Sender: TObject);
2 Var
3   Lista: TList<TLog>;
4   Log: TLog;
5 Begin
6   Lista := TLogDao.Create.Buscar;
7   lbLogDoSistema.Clear;
8   for Log in Lista do
9     lbLogDoSistema.Items.Add(Log.Mensagem);
10 end;
```

Nota

Considere que o código da **Listagem 3** está entre as linhas 25 e 26 da **Listagem 2**.

O código da **Listagem 3** é realmente simples, concorda? Se você não concorda imagine como este código seria mais complicado em um sistema do mundo real se a classe TLogDao não existisse. Na vida real os sistemas normalmente executam dezenas de centenas de ações: seria um caos gerenciar os logs de todas estas ações se não existisse uma única entidade (classe, programador), responsável por isto. Eu acredito que neste ponto você deve estar pensando: "eureca! Entendi a importância do princípio de responsabilidade única". Aparentemente o trabalho está concluído, o que você acha?

Qual é a principal constante de qualquer sistema?

O sistema já estava pronto e funcionando corretamente, até que um cliente pede uma alteração ('alterações', elas chegam **constantemente**). Ele quer uma lista dos logs, como já existe hoje, porém com um adicional: ele deseja que a lista seja agrupada por data. O sistema passa por seu primeiro desafio, pois a implementação de lista existente hoje apenas exibe os logs na ordem em que foram cadastrados (veja **Listagem 3**). Para mudar isto é necessário criar mais um método que agrupe os logs já existentes para depois exibi-los na lista. Porém, antes de fazer o agrupamento dos logs é necessário ordená-los por data para garantir que o agrupamento fique correto. Eis o que vou fazer: veja na **Listagem 4** como é feito o agrupamento dos logs.

A **Listagem 4** mostra os dois novos métodos criados na classe TForm1 parar criar o comportamento desejado por aquele cliente. Na linha 22 você pode ver o método que organiza uma lista de log por ordem de data. O método Ordenar recebe uma lista de logs que será ordenada do log com a data mais antiga para o log com a data mais recente. Ele faz isto usando o algoritmo de ordenação por troca. Mas este não é o método que estamos interessado, pois o cliente pediu para agrupar por data e não para ordenar. Ordenar foi apenas um pré-requisito para poder fazer o agrupamento correto. Entre as linhas 1 e 21 existe a implementação do método ExibirLogAgrupadoPorData que faz exatamente o que o nome sugere. Na linha 8 este método pede uma lista dos logs para um objeto da classe TLogDao e na linha 9 pede para que esta lista seja ordenada. Com a lista ordenada corretamente, é feito um loop por todos os logs da lista, e a cada alteração na data dos logs é adicionada uma separação entre os logs. Assim o objetivo é atingindo e agora o sistema pode exibir uma lista dos logs ordenados e agrupados por data.

É claro, esta não foi uma boa solução. Ela atingiu o objetivo, porém com certeza irá desperdiçar o tempo de qualquer programador em um futuro próximo quando o sistema estiver maior. Por que isto irá acontecer? A primeira coisa a observar é que a classe TForm1 novamente está cheia de responsabilidades. Atualmente ela é responsável por quatro coisas diferentes, sendo que apenas uma delas é interessante para TForm1.

1. Ela controla a interação entre o usuário e as ações do Form1 (este é o objetivo real da classe);
2. Ela exibe um lista simples de todos os logs;
3. Ela exibe uma lista agrupada de todos os logs;
4. Ela ordena uma lista de log.

Listagem 4. Organizando e agrupando os logs do sistema

```
1 procedure TForm1.ExibirLogAgrupadoPorData;
2 Var
3   Logs: TList<TLog>;
4   Log: TLog;
5   UltimoDia, DiaAtual: TDate;
6 begin
7   UltimoDia := 0;//inicia a variavel
8   Logs := TLogDao.Create.Buscar;
9   Ordenar(Logs);
10   lbLogDoSistema.Items.Clear;
11   for Log in Logs do
12   begin
13     DiaAtual := StrToDate(DateToStr(Log.DataHora));
        //Cuidado com este código!
14     if UltimoDia <> DiaAtual then
15     begin
16       UltimoDia := DiaAtual;
17       lbLogDoSistema.Items.Add('Eventos do dia ' + DateToStr(UltimoDia));
18     end;
19     lbLogDoSistema.Items.Add('   ' + Log.Mensagem);
20   end;
21 end;
22 procedure TForm1.Ordenar(out Logs: TList<TLog>);
23 var
24   LogMaior, LogMenor: TLog;
25   i, j: Integer;
26  begin
27   for i := 0 to Logs.Count - 1 do
28   Begin
29     for j := Logs.Count - 1 downto i + 1 do
30     begin
31       LogMaior := Logs.Items[i];
32       LogMenor := Logs.Items[j];
33       if LogMaior.DataHora > LogMenor.DataHora then
34         Logs.Exchange(i, j);
35     end;
36   end;
37 end;
```

Nota

Observe o código estranho da linha 13 da **Listagem 4**. Ele foi uma saída simples pra retirar a parte referente às horas de uma variável do tipo TDateTime. Desta maneira eu consegui uma data pura: uma data que não diz a hora do dia (com isso não foi preciso recorrer à DateUtils).

Nota do DevMan

DAO – Data Access Object é um modelo (ou padrão) muito comum para persistir as informações em banco de dados relacionais. Este modelo visa separar a regra de negócio em uma camada diferente das camadas de acesso a banco de dados. Desta maneira todas as funções relacionadas ao banco de dados, como obter uma conexão ou qualquer outra função que use SQL, deve ficar em classes DAO. No exemplo do artigo a classe TLogDao é uma classe DAO.

Colocar toda esta responsabilidade em uma classe cujo objetivo real é outro – veja item 1 da lista anterior – faz com que todas as demais classes do sistema precisem fazer uma referência a TForm1 para poder usar funções que não são dela. Se isto não for corrigido em pouquíssimo tempo este sistema terá tantas classes interligadas que será praticamente impossível alterar uma delas sem afetar de maneira ruim outra classe qualquer. Para solucionar este problema de forma definitiva e ainda deixar o sistema flexível suficiente para sofrer alterações futuras sem que isto cause um caos, vou aplicar outro princípio de orientação a objetos. Apresento-lhe o princípio 'separe o que varia'. Este princípio diz para separar todo código que varia dos códigos que permanecem iguais. Fazer isto permite que um sistema possa trocar de comportamento sem precisar ser recompilado. Isto é muito interessante! O que você acha? Imagine trocar a maneira como as notas fiscais do seu sistema são impressas sem precisar recompilá-lo. Isto é possível. Vou fazer algo parecido com a lista de logs do sistema de exemplo. Em primeiro lugar é necessário identificar o que está mudando no código deste exemplo. Mas, atenção! Estou falando de mudanças em termos de objetivos. Quais são os trechos de códigos que possuem o mesmo objetivo, porém faz isto de maneiras diferentes? Na classe TForm1 existem dois métodos com o mesmo objetivo: exibir uma lista dos logs. O métodos btExibirLogClick (**Listagem 3**) e ExibirLogAgrupadoPorData (linha 1 da **Listagem 4**) têm o mesmo objetivo, porém fazem isto de maneira diferente. Por isto esta é a parte que varia neste projeto de exemplo. Lembre-se que quando digo variações estou falando de códigos que têm o mesmo objetivo mas variam na maneira como o atingem.

Nota

Conheça o princípio Separe o que Varia. Ele diz que os códigos que atingem o mesmo objetivo de maneira diferente devem ser separados em classes diferentes.

Última chance para entender como separar códigos que variam

O projeto de exemplo precisa ser melhorado. Ele não está flexível! Para melhorá-lo vou separar os trechos de códigos que variam em classes. Para identificar que estas classes têm o mesmo objetivo também será criada uma interface para elas. Não se preocupe, não será nada complicado. Se precisar de mais informações sobre interface veja a os links que indico final deste artigo.

Como eu havia prometido, a **Listagem 5** não mostra nada de complicado. Nela podemos ver uma interface que declara apenas um método: OrganizarListaDeLogs (linha 7). Este método tem o objetivo de organizar uma lista de logs recebida como parâmetro e retornar esta nova lista organizada em um objeto TStrings. Está tudo meio obscuro? Dê uma olhada na **Listagem 6**. Isto pode ajudar a entender melhor.

A **Listagem 6** mostra uma classe que possui apenas um método. Olhe com mais atenção e você verá que a implementação deste método (entre as linhas 12 e 24) é muito parecida com o método btExibirLogClick da **Listagem 3**. Na verdade eles são os mesmos, pois o que eu fiz foi dar um CTRL + X no código da **Listagem 3**, colá-lo no método OrganizarListaDeLogs da **Listagem 6**, além de fazer uma pequena modificação. Foi necessário tirar a parte do código que fazia referência aos componentes do Form1. Veja na linha 9 da **Listagem 3** que cada log do sistema era adicionado no TListBox do Form1, quando na linha 19 da **Listagem 6** cada log é adicionado a uma variável do tipo TStrings declarada na linha 14. Além disso, como já foi dito, é necessário retornar a nova lista – já organizada como resultado da função: isto acontece na linha 20. Estas foram as duas únicas mudanças em relação à **Listagem 3**.

Por trás deste pequeno código, se esconde o princípio de separar o que varia (ou encapsule o que varia). Logo você entenderá por que. Continue e veja o código da **Listagem 7**.

Listagem 5. Interface IListaDeLog

```
1   unit ListaDeLog;
2   Interface
3   uses Log, Generics.Collections;
4   type
5    IListaDeLog = interface
6    ['{D4453126-8B1B-4A73-95CE-0EAB5622E18E}']
7      function OrganizarListaDeLogs(Logs: TList<TLog>): TStrings;
8    end;
9  implementation
10 End.
```

Listagem 6. Implementação de IListaDeLog para listas simples

```
1  unit ListaSimples;
2  Interface
3  uses ListaDeLog, Generics.Collections, Log, dialogs;
4  Type
5    TListaSimples = class(TInterfacedObject, IListaDeLog)
6    Public
7      function OrganizarListaDeLogs(Logs: TList<TLog>): TStrings;
8    end;
9  implementation
10   uses Classes, SysUtils;
11   { TListaSimples }
12   function TListaSimples.OrganizarListaDeLogs(Logs:TList<TLog>):
       TStrings;
13   Var
14    ListaOrganizada: TStrings;
15    Log: TLog;
16 begin
17    ListaOrganizada := TStringList.Create;
18    for Log in Logs do
19      ListaOrganizada.Add(FormatDateTime('DD/MM/YY HH:hh:ss',
        Log.DataHora) + ' => ' + Log.Mensagem);
20    Result := ListaOrganizada;
21 End;
22 End.
```

Repare que o código entre as linhas 14 e 36 da **Listagem 7** é o mesmo apresentado na **Listagem 4**. Até mesmo o código do método Ordenar da **Listagem 4** está presente na **Listagem 7**: veja linha 34. Isto acontece porque estou separando os métodos btExibirLogClick e ExibirLogAgrupadoPorData, da classe TForm1. Em outras palavras, estou separando em duas novas classes as partes da classe TForm1 que variam. Veja na linha 5 das **Listagens 6** e **7** que ambas a classes implementam a mesma interface. Esta interface serve para identificar que estas duas classes têm o mesmo objetivo, apesar de atingi-los de maneira diferente. Você consegue ver o princípio separe o que varia sendo aplicado?

O código deste exemplo já se tornou bem mais flexível, pois agora cada classe do projeto possui apenas um objetivo. Porém, da maneira como está ainda existirá um cruzamento enorme de referências entre as classes. Conforme o sistema for crescendo o número de classes também crescerá. Se todas

estas classes precisarem exibir o log do sistema, teremos um grande número de classes dependentes das classes TListaAgrupada e TListaSimples. Isto não é bom, pois se um destes comportamentos de exibição (uma destas classes de exibição) não for mais necessário, você não consignará excluí-los porque existem outras classes que fazem referências aos arquivos delas. Para diminuir este problema, vou aproveitar da classe TLogDao. Já que o objetivo dela é gerenciar os logs, nada mais justo que ela também saiba organizar a lista destes logs.

Listagem 7. Implementação de IListaDeLog para listas agrupadas

```
1  unit ListaAgrupada;
2  interface
3  uses ListaDeLog, Log, Generics.Collections;
4  Type
5    TListaAgrupada = class(TInterfacedObject, IListaDeLog)
6    private
7      procedure Ordenar(out Logs: TList<TLog>);
8    public
9      function OrganizarListaDeLogs(Logs: TList<TLog>): TStrings;
10   end;
11 implementation
12 uses Classes, SysUtils;
13 { TListaAgrupada }
14 function TListaAgrupada.OrganizarListaDeLogs
    (Logs: TList<TLog>): TStrings;
15 Var
16   ListaOrganizada: TStrings;
17   Log: TLog;
18   UltimoDia, DiaAtual: TDate;
19 Begin
20   Ordenar(Logs);
21   ListaOrganizada := TStringList.Create;
22   UltimoDia := 0;//inicia a variavel
23   for Log in Logs do
24   begin
25     DiaAtual := StrToDate(DateToStr(Log.DataHora));
26     If UltimoDia <> DiaAtual then
27     begin
28       UltimoDia := DiaAtual;
29       ListaOrganizada.Add('Eventos do dia ' + DateToStr(UltimoDia));
30     end;
31     ListaOrganizada.Add('   ' + Log.Mensagem);
32   end;
33   Result := ListaOrganizada;
   end;
34 { Aqui vai o código do método Orderar. Aquele mesmo método
    da listagem 4. }
35 end.
```

TLogDao sofre sua última alteração

A ideia é a seguinte: ao invés de criar referências a estas duas últimas classes criadas no tópico anterior, irei criar um método na classe TLogDao cujo objetivo é usar as duas novas classes para organizar uma lista de log. Desta maneira as outras classes do sistema precisam fazer referência apenas à classe TLogDao. Além disto, ainda existe outra vantagem. O teste (if..then), para saber qual das duas formas de organização de lista devem ser usadas ficará em um único lugar. Isto facilita muito, pois não existirá um monte de if..then espalhado pelo sistema inteiro apenas para testar qual classe de organização será usada. Tudo isto ficará em um único lugar: na classe TLogDao. Tudo que as demais classes do sistema precisarão conhecer é o nome do método de organização da classe TLogDao. Vamos ao código (**Listagem 8**).

A única mudança da classe TLogDao em relação ao que já existia está entre as linhas 20 e 34 da **Listagem 8**. Na linha 20 começa a implementação do método privado ListaSimples. Este método foi a maneira que eu inventei para simular a existência de um parâmetro no sistema onde o usuário pode escolher qual o tipo de organização ele deseja aplicar na lista de logs. Repare que este método retorna aleatoriamente valores TRUE e FALSE. Quando o método ListaSimples retornar TRUE a lista de log será organizada pela classe TListaSimples e quando retornar FALSE a lista será organizada pela classe TListaAgrupada. Por sua vez o método público Organizar, que começa a ser implementado na linha 25, testa o retorno do método ListaSimples e invoca o método OrganizarListaDeLogs da classe correta de acordo com o resultado do método ListaSimples.

Listagem 8. Última alteração da classe TLogDao

```
1  unit Log;
2  Interface
3  uses Generics.Collections, Classes;
4  Type
5    {O record Tlog é declardo aqui }
6    TLogFile = file of TLog;
7    TLogDao = class
8    Private
9      function ArquivoDeLog: string;
10     function ListaSimples: Boolean;
11   Public
12     procedure Salvar(Log: TLog);
13     function Buscar: TList<TLog>;
14     function Organizar(Lista: TList<TLog>): TStrings;
15   end;
16 Implementation
17 uses IOUtils, SysUtils, ListaDeLog, ListaAgrupada, ListaSimples;
18 { TLogDao }
19 { Implementação dos outros métodos. Destes outros métodos no
    exemplo disponível para download }
20 function TlogDao.ListaSimples: Boolean;
21 Begin
22   Randomize;
23   Result := Random(2) = 0;
24 end;
25 function TlogDao.Organizar(Lista: TList<TLog>): TStrings;
26 Var
27   OrganizadorDeLogs: IListaDeLog;
28 Begin
29   if ListaSimples then
30     OrganizadorDeLogs := TListaSimples.Create
31   Else
32     OrganizadorDeLogs := TListaAgrupada.Create;
33   Result := OrganizadorDeLogs.OrganizarListaDeLogs(Lista);
34 end;
35 end.
```

Como fica a classe TForm1 após esta mudança? Certamente fica muito mais simples do que já estava. Veja **Listagem 9**.

Como já foi dito, todos aqueles códigos acrescentados para fazer a organização das listas de logs foram retirados da classe TForm1 (separado da classe TForm1), e no lugar de todas aquelas linhas de código está o método btExibirLogClick implementado na linha 32. Agora a classe TForm1 tem apenas uma responsabilidade: interagir com o usuário através do Form1. Apesar de a classe ter um método para exibir uma lista de log, ela não tem mais o controle sobre como a lista é capturada e muito menos sobre como a lista é organizada. Na verdade a classe TForm1 apenas copia objeto TStrings retornado pelo método Organizar da classe TLogDao para a propriedade Items do TListBox existente no Form1. Já que a classe TForm1 não sabe mais nada a respeito de como os logs são gerados, carregados ou organizados pode-se dizer que o gerenciamento dos logs não é responsabilidade dela.

Listagem 9. O resultado é um código mais simples e puro

```
1  unit Principal;
2  Interface
3  Uses
4    Windows, Messages, SysUtils, Variants, Classes, Graphics,
      Controls, Forms,
5    Dialogs, StdCtrls;
6  Type
7    TForm1 = class(TForm)
8      btAcao1: TButton;
9      btAcao2: TButton;
10     btAcao3: TButton;
11     btExibirLog: TButton;
12     lbLogDoSistema: TListBox;
13     procedure btAcao1Click(Sender: TObject);
14     procedure btAcao2Click(Sender: TObject);
15     procedure btAcao3Click(Sender: TObject);
16     procedure btExibirLogClick(Sender: TObject);
17   Private
18     procedure CriarLog(Mensagem: string);
19     { Private declarations }
20   Public
21     { Public declarations }
22   end;
23 Var
24 Form1: Tform1;
25 Implementation
26 uses Log, Generics.Collections;
27 {$R *.dfm}
28 { Implementação do método btAcao1Click }
29 { Implementação do método btAcao2Click }
30 { Implementação do método btAcao3Click }
31 { Implementação do método CriarLog }
32 procedure TForm1.btExibirLogClick(Sender: TObject);
33 Var
34   Dao: TlogDao;
35   Lista: Tlist<TLog>;
36  Begin
37   Dao := TLogDao.Create;
38   Lista := Dao.Buscar;
39   lbLogDoSistema.Items := Dao.Organizar(Lista);
40 end;
41 end.
```

Conclusão

Você aprendeu neste artigo como organizar os códigos de sua aplicação para obter um sistema com mais qualidade e de carona aprendeu como criar um algoritmo de log para qualquer sistema. Viu um exemplo de como criar códigos mais organizados e consequentemente mais flexíveis usando o princípio de "responsabilidade única" juntamente com o princípio separe o que varia. Existem muitos outros princípios, mas estes dois já causaram impacto suficiente para que o sistema fosse muito mais flexível. A flexibilidade chegou a tal ponto que agora é possível adicionar qualquer outro comportamento de organização de lista sem que seja necessário alterar mais do que um único método no sistema inteiro, mesmo que este sistema tenha centenas de dezenas de classes e métodos. Para adicionar ou remover mais comportamentos de organização é necessário apenas alterar o método Organizar da classe TLogDao.

Agora é sua vez. Vá em frente, baixe o exemplo disponível para download e tente implementar mais comportamentos de organização de lista: que tal exibir uma lista com a quantidade de logs por dia? Tente implementar este novo comportamento. Isto fará com que seu cérebro assimile melhor todo o conceito apresentado neste artigo. Veja na **Figura 2** a versão final do aplicativo e na **Listagem 10** a implementação dos métodos exibidos na **Listagem 1**.

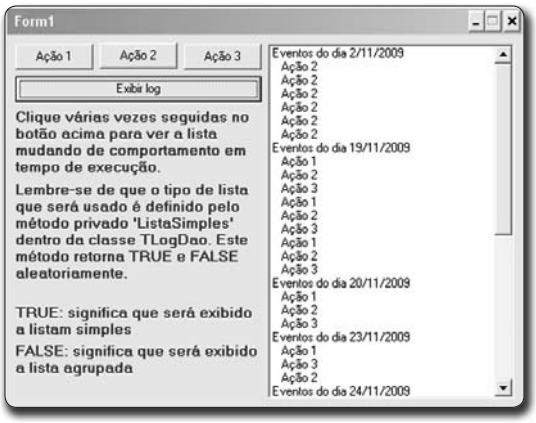


Figura 2. A aplicação em execução

Listagem 10. A parte do código que não aparece na listagem 1

```
1  function TLogDao.ArquivoDeLog: string;
2  Begin
3    Result := TPath.GetDirectoryName(ParamStr(0));
4    Result := Result + '\Logs.log';
5  end;
6  procedure TLogDao.Salvar(Log: TLog);
7  Var
8    LogFile: TLogFile;
9    DataMinima: TDateTime;
10 Begin
11   AssignFile(LogFile, ArquivoDeLog);
12   if TFile.Exists(ArquivoDeLog) then
13   Begin
14     Reset(LogFile);
15     Seek(LogFile, FileSize(LogFile));
16   End
17   Else
18     Rewrite(LogFile);
19   //Eu decidi que um log não pode ter uma data inferior a
      data de ontem
20   DataMinima := Date - 1;
21   if Log.DataHora < DataMinima then
22     Log.DataHora := Now;
23
24   Write(LogFile, Log);
25   CloseFile(LogFile);
26 end;
27 function TLogDao.Buscar: TList<TLog>;
28 var
29   Log: TLog;
30   LogFile: TLogFile;
31 begin
32   AssignFile(LogFile, ArquivoDeLog);
33   Reset(LogFile);
34   Result := TList<TLog>.Create;
35   while not Eof(LogFile) do
36   begin
37     Read(LogFile, Log);
38     Result.Add(Log);
39   end;
40   CloseFile(LogFile);
41 end;
```

Links

Desvendando Generics e Métodos anônimos no Delphi 2009
http://cc.embarcadero.com/Item/26347

Delphi file support
http://www.delphibasics.co.uk/Article.asp?Name=Files

Create your own Database using Delphi's "File Of" Typed Files
http://delphi.about.com/od/fileio/a/fileof_delphi.htm

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Abstract Factory

Flexibilizando a criação de objetos

Giuliano Prado de Morais Giglio
giulianopmg@yahoo.com.br
twitter.com/espacogiu
Desenvolvedor Delphi desde 1997, com ampla experiência em aplicações Win32. Graduado em Informática pela UFJF, com Especialização em Desenvolvimento de Aplicações para Web pelo CES de Juiz de Fora-MG, e Mestrado em Computação pela UFF/Niteroi-RJ. Atualmente é professor universitário em diversas instituições, em cursos de Sistemas de Informação, e atua como consultor, pesquisador e desenvolvedor de aplicações Java, sobretudo na plataforma J2EE para Web, e J2ME, sendo especialista em aplicações Mobile.

Tadeu Moreira de Classe
tadeuneo@oi.com.br
Graduando no Curso Bacharelado em Sistemas de Informações do Centro de Ensino Superior de Juiz de Fora. Técnico em Informática Industrial pelo Colégio Técnico Universitário (CTU / UFJF). Programador Pleno da empresa de softwares jurídicos Novaprolink de Juiz de Fora, MG.

Resumo do DevMan

Na maior parte dos projetos orientados a objetos, quase sempre teremos que lidar com famílias de objetos e a necessidade de criarmos instâncias das inúmeras classes presentes ali. Nem sempre esse processo é bem coordenado e manipulado, sobretudo se considerarmos as mudanças nos requisitos futuros do software. Desta forma, pretendemos apresentar um padrão de projeto que torna essa tarefa bem mais flexível e organizada, a fim de privilegiar a manutenção futura e decompor melhor o problema a ser tratado, promovendo um fraco acoplamento entre as classes e a baixa coesão.

Do que trata o artigo
Vamos entender a importância dos padrões de projeto, mais necessariamente dos padrões criacionais para tratamento de famílias de objetos distintas e a instanciação de seus objetos. Os padrões Factory-Method e Abstract Factory serão implementados para então analisarmos suas vantagens.

Para que serve
Os padrões criacionais permitem uma flexibilização no ato de instanciar objetos pertencentes a uma hierarquia de classes. Isso é obtido através da decomposição do problema para privilegiar uma melhor manutenção do código. Ao fazer isso, conseguimos um fraco acoplamento e uma baixa coesão, que é tudo que um código orientado a objetos necessita.

Em que situação o tema é útil
Auxilia na refatoração de código e manutenções futuras, no caso de novos requisitos impactarem diretamente numa hierarquia de classes.

Um padrão descreve uma solução para um problema que ocorre com frequência durante o desenvolvimento de software, podendo ser considerado como um par "problema/solução" (Buschmann, 2003). Projetistas familiarizados com certos padrões podem aplicá-los imediatamente a problemas de projeto, sem ter que redescobri-los (Gamma, 2000). Um padrão é um conjunto de informações instrutivas que possui um nome e que capta a estrutura essencial e

o raciocínio de uma família de soluções comprovadamente bem sucedidas para um problema repetido que ocorre sob um determinado contexto e um conjunto de repercussões (Appleton, 2000).

Padrões de software podem se referir a diferentes níveis de abstração no desenvolvimento de sistemas orientados a objetos. Assim, existem padrões arquiteturais, em que o nível de abstração é bastante alto, padrões de análise, padrões de projeto, padrões de código, entre outros. As diversas categorias de padrões são discutidas a seguir. O uso de padrões proporciona um vocabulário comum para a comunicação entre projetistas, criando abstrações num nível superior ao de classes e garantindo uniformidade na estrutura do software (Gall,1996). Além disto, eles atuam como blocos construtivos a partir dos quais, projetos mais complexos podem ser construídos (Gamma, 2000).

Diversos autores têm proposto centenas de padrões nas mais diversas áreas de aplicação. Esses padrões são descobertos após a abstração de fatores comuns em diversos pares problema-solução, podendo-se situar em várias faixas de escala e abstração. Existem padrões de domínio específico, como por exemplo, para sistemas multimídia educativos e também padrões independentes de domínio, como por exemplo padrões para projeto de interface. Há padrões que ajudam a estruturar um sistema de software em subsistemas e outros que ajudam a implementar aspectos de projeto particulares.

Olhando mais de perto para muitos padrões, percebe-se que um padrão resolve um problema, mas sua aplicação pode gerar outros problemas, que podem ser resolvidos por outros padrões. Em geral não existem padrões isolados: um padrão pode depender de outro no qual esteja contido, ou das partes que ele contém, ou ser uma variação de outro, ou ser uma combinação de outros. De maneira geral, um padrão e seus variantes descrevem soluções para problemas muito similares, que variam em algumas das influências envolvidas.

Dentre os inúmeros padrões de projeto, apresentaremos neste artigo o Abstract Factory, classificado na categoria dos de Criação (Horstmann, 2007), os quais são responsáveis por abstrair o processo de criação de objetos a partir de instanciação de classes. Trata-se de um padrão extremamente útil no trato da instanciação de famílias de objetos. Para que sua implementação seja viabilizada, utilizaremos também outro padrão chamado de Factory Method importante principalmente na instanciação de objetos numa hierarquia de classes, sem que a variação de seus participantes influencie no seu processo.

Conhecendo Factory Method

Todos os padrões de criação lidam com a melhor forma de se criar instâncias dos objetos. Isto é importante porque seu programa não deveria depender de como os objetos são criados e arranjados. Em Delphi, a maneira mais simples de se criar um objeto é usando o método Create:

```
oMeuObjeto := TminhaClasse.create;
```

No entanto, isto pode tornar a criação de objetos no seu programa um tanto quanto inflexível. Em muitos casos, a natureza dos objetos que são criados pode variar com as necessidades do programa e a abstração do processo de criação de objetos para uma classe especial de criação pode fazer seu programa mais flexível.

Por exemplo, é possível criar um objeto sem ter conhecimento algum de sua classe concreta? Imaginemos uma aplicação principal, a qual chamaremos de aplicação Cliente, podendo ser um Formulário ou módulo principal do sistema, somente para ilustrar. Imaginemos que desejamos instanciar um objeto da hierarquia de classes a seguir (**Figura 1**).

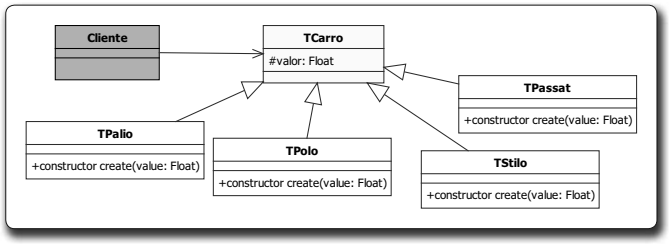


Figura 1. Hierarquia de classes sobre Carros

Provavelmente a aplicação Cliente teria que conhecer as classes concretas que participam dessa hierarquia (**Listagem 1**).

Listagem 1. Instanciação de objetos

```
Var
        oPalio: TPalio;
        oPolo: TPolo;
        oStilo: TStilo;
        oPassat: TPassat;
Begin
        oPalio := TPalio.create;
        ....
        oPolo := TPolo.create;
        ....
End;
```

Mesmo se utilizássemos Polimorfismo, não eximiríamos a aplicação Cliente se referenciar a tais classes. O problema se torna significativo quando ocorrer uma variação das subclasses participantes, ou seja, caso outras classes sejam herdadas e participem da família ou mesmo trocadas pelas originais, o quanto isso afetaria o processo de manutenção no código da aplicação Cliente? Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no Cliente. Desta forma, a aplicação do padrão de projeto Factory Method, permite com que encapsulemos a forma como a instanciação dos objetos é feita, retornando o objeto desejado para a aplicação Cliente, sem que a variação das subclasses modifique o código daquele, tornando-o independente da hierarquia proposta. Portanto, o Factory Method define uma interface comum para criar objetos, sendo que o objeto específico é determinado nas diferentes implementações dessa interface.

No caso da **Figura 1**, temos a hierarquia implementada da seguinte forma: TCarro : super-classe da hierarquia, contendo somente um atributo private valor e seus métodos de acesso Get/Set. Veja **Listagem 2**.

As classes TPalio, TPolo, TStilo, TPassat são as subclasses de TCarro, contendo somente a redefinição do construtor, com inicialização distinta do atributo valor para cada classe.

Sendo assim, na **Listagem 3**, escolhemos a classe TPalio para demonstrar a sua implementação completa.

Listagem 2. Implementação da superclasse TCarro

```
unit uCarro;

interface

type
  TCarro = class
  private
    Valor : double;
  public
    function GetValor : double;
   procedure SetValor(Value : double);
  end;

implementation

{ TCarro }

function TCarro.GetValor: double;
begin
  Result := Valor;
end;

procedure TCarro.SetValor(Value: double);
begin
  Valor := Value;
end;

end.
```

As demais classes seguirão o padrão da classe TPalio, modificando somente o valor de inicialização do atributo valor de cada classe, como observado na **Listagem 4**.

Para a implementação do Factory Method, criamos uma classe de nome TCarroFactory. Definimos o método GetCarro o qual é o Factory Method propriamente dito, onde será passado como parâmetro o nome do carro que o usuário desejar. O método retornará uma instância de uma das classes da hierarquia, definido pelo valor do parâmetro passado, como mostrado na **Listagem 5**. Desta forma, esta classe agora será responsável pelas instâncias dos carros informados pelos usuários.

Nossa implementação terá ao final a seguinte estrutura no modelo anteriormente proposto, segundo a **Figura 2**.

Para testar o padrão, vamos criar nosso Cliente (um formulário) o qual chamamos de frmAbstractFacotory e a sua unit de FabstractFactory. Utilizamos dentro do formulário um GroupBox e o nomeamos de gpbCarro. Dentro dele colocamos um ComboBox e o nomeamos de cmbCarro, definindo a sua propriedade Items com os nomes dos carros de sua hierarquia (Palio, Polo, Stilo, Passat). Ainda dentro de gpbCarro, colocamos um Label e mudamos a sua propriedade Caption para "Valor R$:", além de um Edit como o name de edtValor.

Listagem 3. Implementação da subclasse TPalio

```
unit uPalio;

interface

uses uCarro;

type
  TPalio = class(TCarro)
    public
      constructor Create;
  end;

implementation

{ TPalio }

constructor TPalio.Create;
begin
  Self.Valor := 32000;
end;

end.
```

Listagem 4. Implementação das subclasses restantes da hierarquia

```
{ TPolo }

constructor TPolo.Create;
begin
  Self.Valor := 40000;
end;


{ TStilo }

constructor TStilo.Create;
begin
  Self.Valor := 53000;
end;


{ TPassat }

constructor TPassat.Create;
begin
  Self.Valor := 71000;
end;
```

No código de frmAbstractFacotory, referencie as units uCarroFactory e uCarro para que elas possam ser usadas dentro deste formulário, como indicado na **Figura 3**.

```
unit FAbstractFactory;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, uCarroFactory, uCarro, uFabricante;
```

Figura 3. Referência de uCarroFactory e uCarro

No evento OnSelect do ComboBox cmbCarro, estaremos testando o uso do Factory Method, como mostrado na **Listagem 6.**

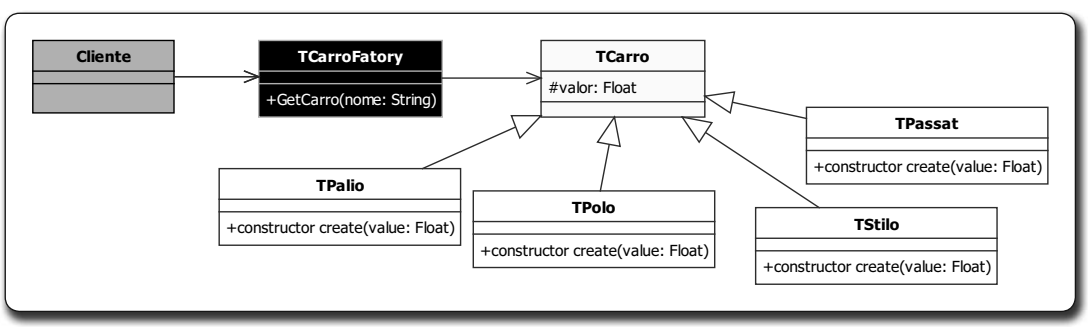


Figura 2. Modelo de classes com a aplicação do Factory Method

Listagem 5. Implementação da classe TCarroFactory

```
unit uCarroFactory;

interface

uses
  uCarro, uPassat, uPalio, uPolo, uStilo;

type
  TCarroFactory = class
    public
      class function GetCarro(aNomeCarro : string) : TCarro;
  end;

implementation
{ TCarroFactory }

class function TCarroFactory.GetCarro(aNomeCarro: string):
TCarro;
begin
  Result := nil;

  if (aNomeCarro = 'Palio') then begin
    Result := TPalio.Create;
  end
  else if (aNomeCarro = 'Polo') then begin
    Result := TPolo.Create;
  end
  else if (aNomeCarro = 'Stilo') then begin
    Result := TStilo.Create;
  end
  else if (aNomeCarro = 'Passat') then begin
    Result := TPassat.Create;
  end;
end;

end.
```

Listagem 6. Implementação do evento OnSelect de cmbCarro

```
procedure TfrmAbstractFacotory.cmbCarroSelect(Sender: TObject);
var
  vCarro : TCarro;
begin
  vCarro := TCarroFactory.GetCarro(cmbCarro.Text);
  edtValor.Text := FormatFloat('0.00', vCarro.Valor);
end;
```

Ao selecionar o item correspondente a um carro, o passamos para GetCarro que instancia o carro relacionado, mostrando no edit edtValor o seu valor correspondente, como vimos na **Figura 4**.

Figura 4. Formulário de teste do padrão Factory Method

Esse é o princípio fundamental do padrão Factory Method. Você cria uma abstração, na qual se decide qual das opções de classe retornar. Então você apenas chama o método comum desta classe, sem sequer saber ou se preocupar qual classe você está realmente lhe oferecendo. Desta forma, TCarroFactory torna-se uma classe chamada de fábrica.

Implementando Abstract Factory

O padrão Abstract Factory está a um nível de abstração maior do que o padrão Factory Method. Você pode usar este padrão quando deseja retornar uma das muitas classes de objetos relacionadas, cada qual pode retornar diferentes objetos se requisitado. Em outras palavras, o padrão Abstract Factory é uma fábrica de objetos que retorna uma das várias fábricas.

Uma aplicação clássica do Abstract Factory é o caso onde o seu sistema precisa de suporte a múltiplos tipos de interfaces gráficas, como Windows, Motif ou MacIntosh. Você diz ao factory que quer que seu programa se pareça com o Windows e ele retorna a fábrica GUI que retorna os objetos relativos ao Windows. Então, quando você requisita um objeto específico como um botão, checkbox's e janelas, a fábrica de GUI retorna as instâncias desses componentes para o Windows.

Para vermos na prática e entendermos o Abstract Factory, vamos utilizar o exemplo parecido com o anterior, porém estendido. Veja o modelo da **Figura 5**.

Temos duas classes TFiat e TVolkswagen, onde através delas dividimos os carros pela sua marca, ou seja, os carros da marca Fiat e Volkswagen. Seguindo a ideia implementada anteriormente, possuímos agora duas fábricas de objetos, cada qual possuindo a responsabilidade de instanciar seus próprios objetos.

O objetivo aqui é aplicarmos o padrão Abstract Factory para realizarmos a instanciação em duas etapas: primeiro instanciarmos a fábrica para, a partir dela, instanciarmos um carro específico da mesma. Mantemos aqui também o encapsulamento das classes, ou seja, o Cliente não saberá quais fábricas estão disponíveis nem quais objetos estas possuem, mantendo o fraco acoplamento na solução adotada.

Vamos iniciar criando a classe TFabricante, que será minha fábrica abstrata. Como atributos coloquemos uma variável oMarca do tipo String como protected e dentro de public vamos fazer as funções de classe GetCarro e GetFabrica, sendo este primeiro um método abstrato (**Listagem 7**).

Listagem 7. Interface da classe TFabricante

```
unit uFabricante;

interface

uses uCarro;

type
  TFabricante = class
  protected
    oNome : string;
  public
    class function GetCarro(aNomeCarro : string) : TCarro;  virtual;

    abstract;
    class function GetFabrica(aNomeFabrica : string) : TFabricante;
  end;
```

Não iremos implementar os métodos desta classe neste momento, serão mostrados na **Listagem 10**. O método GetCarro é abstrato e, portanto, sua implementação será feita nas sub-classes, no caso TFiat e TVolkswagen. Isto é fundamental porque, de fato, queremos que cada fábrica saiba quais as classes que deverá ser encarregada de instanciar, como vemos na implementação da classe fábrica TFiat na **Listagem 8.**

A implementação da classe TVolkswagen é análoga a anterior, modificando logicamente os objetos que GetCarro retornar, **Listagem 9**.

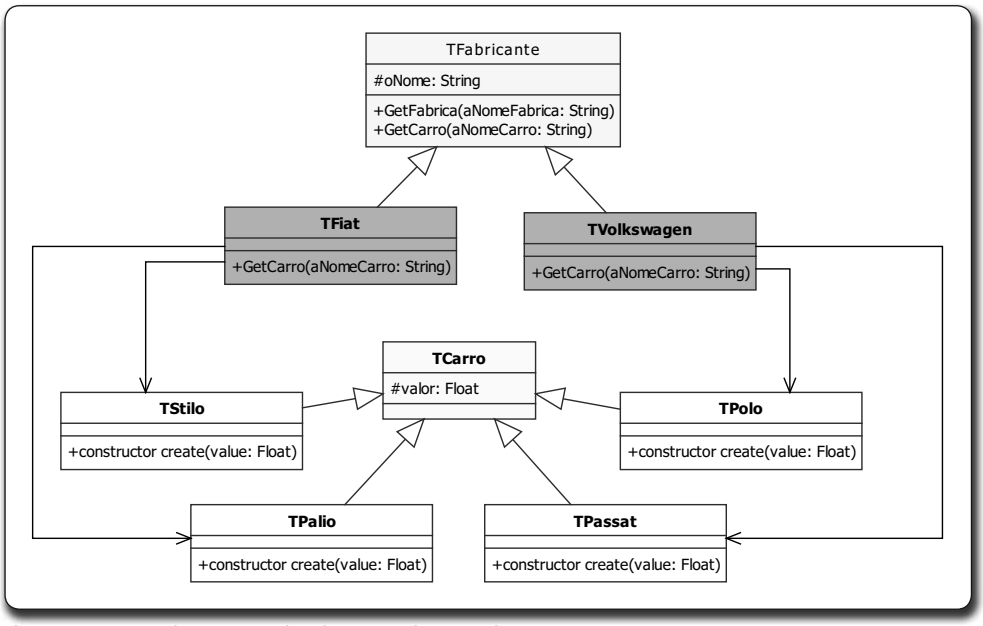


Figura 5. Estrutura hierárquica das classes implementadas

Para finalizarmos esta hierarquia de fábricas, voltemos a classe TFabricante. Referencie as units uVolkswagen e uFiat na uses da seção implementation, para que não haja referência cíclica de units, ou seja, a superclasse referenciando as classes filhas e as classes filhas referenciando a superclasse. Implementemos o método GetFabrica, cuja função é criar objetos das fábricas escolhidas sendo passado por parâmetro o nome de alguma delas, veja **Listagem 10**.

Com todas as nossas hierarquias de classes prontas, podemos fazer um exemplo de uso do padrão, utilizando o próprio formulário frmAbstractFacotory. Colocamos outro GroupBox com o nome de gpbFabrica. Dentro de gpbFabrica, colocamos duas ComboBox, cmbFabrica e cmbCarroFab. Na propriedade Items de cmbFabrica, informamos o nome das nossas fábricas, "Volkswagen" e "Fiat", e na propriedade Items de cmbCarroFab, "Polo, Passat, Palio e Stilo". Inserimos também um Edit, colocando o nome de edtValorFab, onde será exibido o preço do carro selecionado. Ainda dentro de gpbFabrica adicionamos um botão, trocando o seu Caption para "Buscar Carro" e seu nome para "btnFabrica". No evento OnClick de btnFabrica, implementamos com o seguinte código da **Listagem 11**.

Foram declaradas duas variáveis, vFabricante que conterá uma instância de TFabricante e vCarro, que conterá uma instância de um carro da fábrica. A instância de TFabricante é obtida através da função GetFabrica, onde é passado o texto selecionado de cmbFabrica. Após a seleção da fábrica (e se houver algum objeto da mesma), obtemos o carro usando a função GetCarro da recém criada instância vFabricante, passando como parâmetro para esta função o nome do carro selecionado em cmbCarroFab. Com o objeto de carro instanciado, podemos informar o preço do mesmo, acessando o atributo Valor de vCarro, mostrando-o no edit edtValorFab, como mostramos na **Figura 6**.

Listagem 8. Implementação da classe TFiat

```
unit uFiat;

interface

uses
  uCarro, uFabricante, uPalio, uStilo;

type
  TFiat = class(TFabricante)
   constructor Create;
   class function GetCarro(aNomeCarro : string) : TCarro; override;
  end;


implementation

{ TFiat }

constructor TFiat.Create;
begin
  oMarca := 'Fiat';
end;

class function TFiat.GetCarro(aNomeCarro: string): TCarro;
begin
  Result := nil;
  if (aNomeCarro = 'Palio') then begin
    Result := TPalio.Create;
  end
  else if (aNomeCarro = 'Stilo') then begin
    Result := TStilo.Create;
    end;
 end;

end.
```

Listagem 9. Implementação da classe TVolkswagen

```
implementation

{ TVolkswagen }

constructor TVolkswagen.Create;
begin
  oMarca := 'Volkswagen';
end;

class function TVolkswagen.GetCarro(aNomeCarro: string): TCarro;
begin
  Result := nil;
  if (aNomeCarro = 'Polo') then begin
    Result := TPolo.Create;
  end
  else if (aNomeCarro = 'Passat') then begin
    Result := TPassat.Create;
  end;
end;
```

Listagem 10. Implementação do método GetFabrica da classe TFabricante

```
implementation

uses uVolkswagen, uFiat;

{ TFabricante }

class function TFabricante.GetFabrica(aNomeFabrica: string):
TFabricante;
begin
  Result := nil;
  if (aNomeFabrica = 'Volkswagen') then begin
    Result := TVolkswagen.Create;
  end
  else if (aNomeFabrica = 'Fiat') then begin
    Result := TFiat.Create;
  end;
end;

end.
```

Listagem 11. Implementação do evento OnClick do botão btnFabrica

```
procedure TfrmAbstractFacotory.btnFabricaClick(Sender: TObject);
var
  vFabricante: TFabricante;
  vCarro: TCarro;
begin
  vFabricante := TFabricante.GetFabrica(cmbFabrica.Text);
  if (vFabricante = nil) then begin
    ShowMessage('Fábrica não encontrada!');
    Exit;
  end;
  vCarro := vFabricante.GetCarro(cmbCarroFab.Text);
  if (vCarro = nil) then begin
    ShowMessage('Carro não encontrado na fábrica!');
    Exit;
  end;
  edtValorFab.Text := FormatFloat('0.00', vCarro.Valor);
end;
```

Figura 6. Formulário de teste do padrão Abstract Factory

Conclusão

Assim fica finalizado o nosso exemplo do Padrão de Projeto Abstract Factory. As hierarquias aqui foram feitas para exemplificar o padrão, sendo que é possível fazer classes hierárquicas muito maiores.

O desenvolvimento de um software pode ser considerado um trabalho artesanal. Cada profissional soluciona os problemas que encontra usando seu conhecimento e criatividade. Com o uso de padrões isto sofre alterações, pois boas soluções passam a ser documentadas, transmitidas e adotadas por outros profissionais. Como consequência é intensificada a criação de uma linguagem comum para a comunicação e troca de experiências entre os profissionais.

A utilização de padrões também é uma forma eficaz de difusão do conhecimento, pois através do estudo de padrões encontrados em livros e em páginas da Internet pode-se aprender com a experiência de especialistas do mundo inteiro boas práticas de desenvolvimento de software.

Dentre os vários padrões catalogados, procuramos aqui apresentar o Abstract Factory, juntamente com o Factory Method os quais são importantes para se flexibilizar a instanciação de objetos numa herança de classes, chamada também de família de objetos. ●

Nota do DevMan

Os modelos de classes feitos foram implementados pela ferramenta StarUml (http://staruml.sourceforge.net/en/index.php), sendo um projeto Open Source para o desenvolvimento rápido e flexível de diagramas dos mais diversos tipos, que utilizam o padrão UML/MDA (Model Driven Architecture). Seu desenvolvimento se restringe à plataforma Windows e, para atender às exigências de funcionalidades dos profissionais, a ferramenta fornece uma arquitetura simples e poderosa, na qual o usuário pode desenvolver os projetos voltados a linguagens de programação compatíveis (C++, Delphi, C#, VB). Podemos citar outros recursos fornecidos, tais como:

* Implementa praticamente todos os diagramas da UML 2.0 atual (Classes, Casos de Uso, Atividade, Colaboração, Estado, Sequência, Objetos, Componentes);
* Engenharia Reversa total e parcial, dependendo da linguagem;
* Geração de Documentação em formatos como PDF, HTML e de forma organizada;
* Facilidade de instalação e portabilidade entras as diferentes versões da ferramenta (o que é uma restrição em outras ferramentas do mercado);
* Compreende todo o processo de modelagem da análise OO até a geração de código, nas linguagens supracitadas (falha em algumas também).

Referências Bibliográficas

(Appleton, 2000) Appleton, B. Patterns and Software: Essential Concepts and Terminology, disponível em URL: http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html

(Buschmann, 2003) Buschmann, F. et at. Pattern-Oriented Software Architecture, Volume 1, A System of Patterns, Wiley, 2003.

(Fowler, 2004) Fowler, M.; Refatoração - Aperfeiçoando o projeto de código existente. Editora Bookman, 2004.

(Gall, 1996) Gall, H.C., Klösh, R.; Mittermeir, R.T. Application Patterns in Re-Engineering: Identifying and Using Reusable Concepts. Proceedings of the 6th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, Julho 1996, p. 1099-1106.

(Gamma, 2000) Gamma et. Al. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Editora Bookman, 2000.

(Horstmann, 2007) Horstmann, C. Padrões e Projeto Orientados a Objetos. Ed. Bookman, 2ª edição, 2007.

(Shalloway, 2004) Shalloway, A.; Trott, J.R. Explicando Padrões de Projeto: Uma Nova Perspectiva em Projeto Orientado a Objetos. Editora Bookman, 2004.

Links

Introdution to Design Patterns in Delphi
http://www.obsof.com/delphi_tips/pattern.html

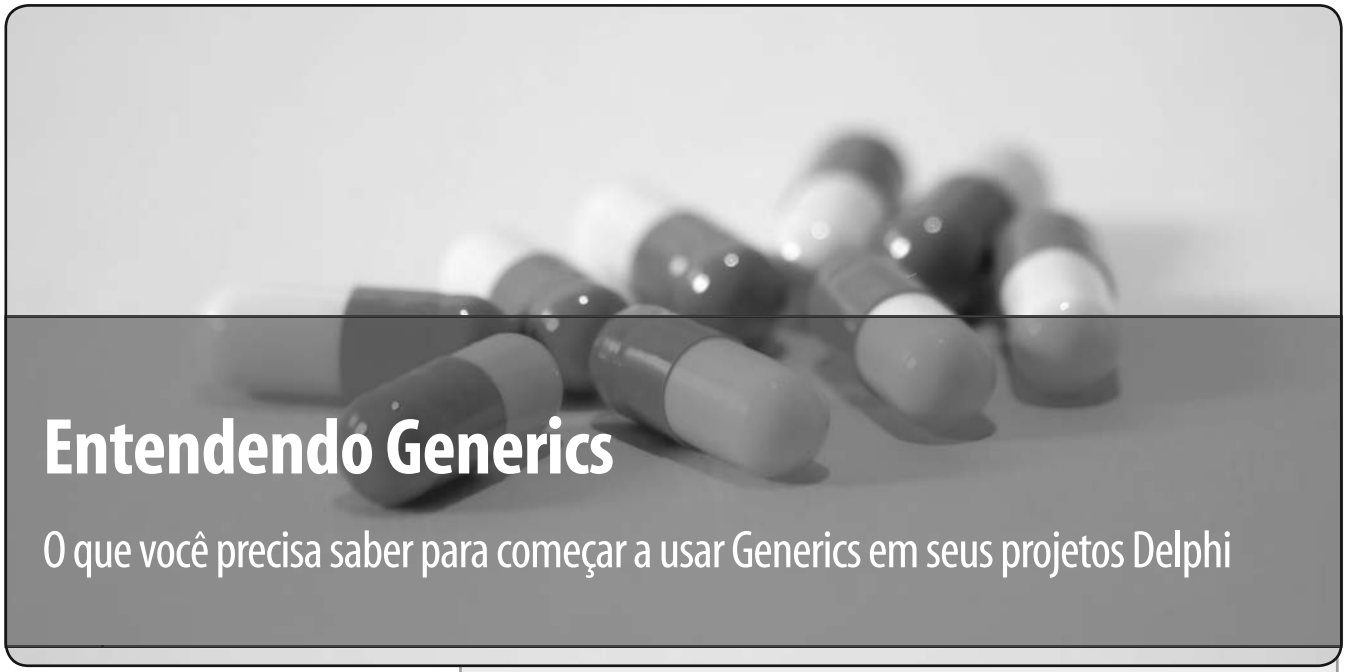
The Gang of Four Design Patterns in Delphi
http://www.felix-colibri.com/papers/design_patterns/gof_design_patterns/gof_design_patterns.html

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos para iniciantes na linguagem Delphi



Entendendo Generics

O que você precisa saber para começar a usar Generics em seus projetos Delphi

Do que trata o artigo

Nesse artigo abordaremos Generics, tanto os tipos já existentes no Delphi como a criação de nossos próprios tipos. Criaremos métodos genéricos, classes genéricas e coleções genéricas. Também mostraremos como podemos tornar nossas aplicações mais flexíveis, porém coerentes utilizando constraints.

Para que serve

Generics é um novo recurso do Delphi que permite que criemos tipos parametrizados. Com tipos parametrizados queremos dizer que um parâmetro que passamos para nossa classe ou método será substituído por um tipo. Assim como os parâmetros de um método são substituídos por valores ou endereços de variáveis, o parâmetro genérico é substituído por um tipo de dado, podendo ser um tipo primitivo, estruturado ou uma classe.

Em que situação o tema é útil

Sempre que precisamos declarar um atributo, método ou variável em uma classe, mas não sabemos de que tipo criar, ou o tipo deve ser variável em tempo de compilação. Não se aplica, porém, nos casos em que o tipo é usado dentro da própria classe ou alterado em tempo de execução. Para isso usamos interfaces. Usando Generics podemos poupar linhas de código e reaproveitar muitas linhas, porque não criamos uma lista infindável de classes irmãs, quase iguais, para trabalhar com tipos de dados diferentes. Também não criamos uma longa lista de métodos sobrecarregados simplesmente para mudar o tipo dos argumentos ou retornos.

Resumo do DevMan

Explicaremos de maneira direta e com exemplos como usar a nova sintaxe do Delphi para tipos parametrizados, conhecidos como Generics. Mostraremos como usar a sintaxe, como criar seus próprios tipos parametrizados, como criar métodos parametrizados e como um parâmetro genérico interfere no próprio funcionamento da classe em si. Mostraremos também alguns tipos genéricos do próprio Delphi e algumas maneiras práticas de se aplicar o que aprendemos. Por último, falaremos sobre como limitar o tipo genérico com constraints, tanto para não permitir que qualquer tipo de dado seja usado como para fornecer à classe alguma informação básica sobre o parâmetro genérico.

Vitor Luiz Rubio

MCP, Analista de Sistemas Sr. Na Editora Revista dos Tribunais. Trabalha com Delphi desde a versão 3. Formado em Processamento de Dados pela FATEC-SP.

Um dos novos recursos do Delphi são Generics. Eles são similares aos templates do C++, mas não totalmente idênticos, por isso não vamos compará-los. São mais parecidos com os Generics do C#, então quem já está familiarizado com o ambiente .NET não sentirá dificuldades em usá-los no Delphi, e ainda ficará feliz em saber que agora podemos desfrutar desse recurso em um compilador de executáveis Win32 nativos.

Generics é um tipo de dado flexível. São tipos parametrizados, ou, mais claramente, parâmetros-tipo, que podem ser definidos em design. Não devemos confundir os parâmetros genéricos com os tipos propriamente ditos. Generics não são tipos, são parâmetros. Assim como os parâmetros ou argumentos de um método são substituídos por um valor, o parâmetro do tipo genérico é substituído por um tipo. Os Generics fornecem uma nova forma de flexibilização dos nossos sistemas, para somar-se às formas que nós já conhecemos, como interfaces, padrões de projeto e RTTI.

Podemos criar métodos com argumentos cujos tipos podem ser indefinidos, assim, podemos trocar o tipo de dado que o método usa cada vez que precisarmos, sem ter a obrigação de criar vários métodos diferentes. Também é possível criar classes que podem virtualmente se conectar com quaisquer outras, porque contém métodos ou propriedades que aceitam dados de qualquer tipo.

Você pode argumentar que sempre fez isso usando Variants quando o tipo de dado era um tipo primitivo, ou um objeto da classe TObject quando o tipo fosse uma classe qualquer. Porém, nesses casos, você era obrigado a usar os comandos da RTTI "is" e "as" para tentar fazer um TypeCast seguro, ou disparar uma exceção caso o tipo não fosse o correto. Nem sempre você conseguiria o tipo correto e isso era causa de erros no seu código.

Outras vezes, para flexibilizar a passagem de parâmetros entre os objetos, porém limitando para que não fosse usado qualquer tipo, nós usaríamos interfaces. Fazendo com que vários objetos implementassem a mesma interface você poderia intercambiá-los à vontade, ainda com a vantagem de acesso a todos os métodos da interface, do AutoComplete em tempo de design, no IDE, e da notificação de algum erro na hora da compilação.

Então, para que Generics?

Generics permitem que o tipo de dado passado seja um tipo primitivo, como Integer ou Double, ou um objeto de uma classe, ou ainda um tipo estruturado como um Record. Além disso, caso você informar o tipo errado, você receberá a mensagem de erro na hora da compilação, e não em tempo de execução, com o programa em produção, causando um Access Violation. E ainda tem a vantagem de ter acesso a todas as informações do objeto em tempo de Design, através do AutoComplete.

Na maioria dos casos usam-se os tipos genéricos para se compor listas de objetos, ou objetos que contêm outros. O Delphi já vem com algumas classes em suas bibliotecas que usam Generics, mais adiante falaremos delas.

Primeiro exemplo com Generics

O primeiro exemplo mostra como é a sintaxe para se criar um método genérico. Vamos criar um novo projeto VCL Forms para Win32 e dar o nome de ConhecendoGenerics.dpr ao projeto e frmGenerics para a Form principal. Criaremos na unit principal duas classes: TPessoa e TOperacao. A classe pessoa terá apenas a propriedade Nome e o método DigaOla. A classe TOperacao terá um método para trocar dois objetos do tipo TPessoa. Vamos adicionar um botão no formulário, chamado btEx1 com o Caption "Exemplo 1" para executar as operações. A classe TPessoa terá um construtor especial onde poderemos inicializar sua propriedade Nome. Então o nome da pessoa poderá ser alterado tanto ao se instanciar o objeto quanto ao mudar sua propriedade Nome. O construtor especial marcamos como overload porque, como a classe é descendente de Tobject, ela já possui um construtor Create implícito, sem argumentos, e nós não queremos substituí-lo, mas queremos ficar com os dois. Vejamos os código da **Listagem 1.**

Listagem 1. Classes TPessoa e TOperacao

```
TPessoa = class
private
  FNome: string;
public
  procedure DigaAlo;
  constructor Create(umNome: string);  overload;
  property Nome: string read FNome write FNome;
end;

TOperacao = class
public
  procedure Troca<TipoDeDado>(var Esquerda, Direita: TipoDeDado);
end;

{...}

implementation

{ TPessoa }
constructor TPessoa.Create(umNome: string);
begin
  FNome := umNome;
end;

procedure TPessoa.DigaAlo;
begin
  ShowMessage('Olá, meu nome é ' + FNome);
end;

{ TOperacao }
procedure TOperacao.Troca<TipoDeDado>(var Esquerda, Direita:
 TipoDeDado);
var
  Temp: TipoDeDado;
begin
  Temp := Esquerda;
  Esquerda := Direita;
  Direita := Temp;
end;
```

Criamos duas classes absolutamente sem nenhum segredo. Porém, repare que entre os sinais de menor e maior pusemos o identificador TipoDeDado. Este é um parâmetro genérico e será substituído pelo tipo que você quiser mais a frente. Os dois argumentos do método Troca são definidos como sendo do tipo TipoDeDado. Com isso temos um tipo dinâmico. O código da **Listagem 2** mostra como podemos usar isso. Esse código basicamente instancia dois objetos do tipo TPessoa, troca-os e os apresenta novamente. Depois repetimos o teste com outros tipos de dados.

Listagem 2. Evento OnClick do botão

```
procedure TfrmGenerics.btEx1Click(Sender: TObject);
var
  Pessoa1, Pessoa2: TPessoa;
  Numero1, Numero2: integer;
  Data1, Data2: TDateTime;
  op: TOperacao;
begin
  Pessoa1 := TPessoa.Create;
  Pessoa2 := TPessoa.Create;
  op := TOperacao.Create;
  Pessoa1.Nome := 'Blaise Pascal';
  Pessoa2.Nome := 'Albert Einstein';
  Pessoa1.DigaAlo;
  Pessoa2.DigaAlo;
  ShowMessage('Trocando...');
  op.Troca<TPessoa>(Pessoa1, Pessoa2);
  Pessoa1.DigaAlo;
  Pessoa2.DigaAlo;
  Numero1 := 1;
  Numero2 := 2;
  op.Troca<integer>(Numero1, Numero2);
  ShowMessage(IntToStr(Numero1));
  ShowMessage(IntToStr(Numero2));
  Data1 := StrToDate('14/02/1983');
  Data2 := Now;
  op.Troca<TDateTime>(Data1, Data2);
  ShowMessage(FormatDateTime('dd/mm/yyyy', Data1));
  ShowMessage(FormatDateTime('dd/mm/yyyy', Data2));
  Pessoa1.Free;
  Pessoa2.Free;
  op.Free;
end;
```

Colocamos nesse primeiro exemplo, além de duas pessoas, números inteiros e datas, mostrando-os com mensagens, para demonstrar que o parâmetro TipoDeDado pode ser substituído por qualquer coisa, inclusive tipos primitivos. Usamos a função IntToStr para transformar o número inteiro em String, para mostrarmos com ShowMessage e fizemos a mesma coisa transformando Data em String com a função FormatDateTime.

Com isso você não precisa criar um descendente de TOperacao para cada tipo de dado que deseja "trocar", nem criar infindáveis métodos Troca com overloads, onde muda-se somente o tipo do parâmetro. Além disso, se o tipo fosse TObject, você correria o risco de passar o tipo errado, mas só perceber com o programa em produção. Usando Generics você não corre esse risco. Para ilustrar, vamos criar mais uma classe chamada TCachorro, com a propriedade Nome e o método Late, conforme podemos ver na **Listagem 3**.

Listagem 3. Classe TCachorro

```
  TCachorro = class
  private
    FNome: string;
  public
    procedure Late;
    constructor Create(umNome: string);  overload;
    property Nome: string read FNome write FNome;
  end;

{...}

implementation

{ TCachorro }

constructor TCachorro.Create(umNome: string);
begin
  FNome := umNome;
end;

procedure TCachorro.Late;
begin
  ShowMessage('rrrrrr au, au!');
end;
```

Criaremos agora mais um método na classe TOperacao para trocar sem usar Generics, à maneira antiga, usando TObject, a nossa classe TOperacao ficará como na **Listagem 4** após as alterações.

Listagem 4. Alterações na classe TOperacao

```
  TOperacao = class
  public
    procedure Troca<TipoDeDado>(var Esquerda, Direita:
     TipoDeDado);   overload;
    procedure Troca(var Esquerda, Direita: TObject);  overload;
  end;

implementation

{ TOperacao }
procedure TOperacao.Troca(var Esquerda, Direita: TObject);
var
  Temp: TObject;
begin
  Temp := Esquerda;
  Esquerda := Direita;
  Direita := Temp;
end;

procedure TOperacao.Troca<TipoDeDado>(var Esquerda, Direita:
  TipoDeDado);
var
  Temp: TipoDeDado;
begin
  Temp := Esquerda;
  Esquerda := Direita;
  Direita := Temp;
end;
```

Adicionaremos no nosso formulário mais um botão chamado btEx2 com o Caption "Exemplo 2". No código desse botão faremos várias experiências de troca. O código pode ser visto na **Listagem 5**.

Listagem 5. Evento OnClick do segundo botão

```
procedure TfrmGenerics.btEx2Click(Sender: TObject);
var
  Pessoa: TObject;
  Cachorro: TObject;
  op: TOperacao;
begin
  Pessoa  := TPessoa.Create('Vitor Rubio');
  Cachorro:= TCachorro.Create('Cujo');
  op := TOperacao.Create;
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  //op.Troca<TPessoa>(Pessoa, Cachorro); //Impossível compilar
  //op.Troca<TCachorro>(Pessoa, Cachorro); //Impossível compilar
  op.Troca(Pessoa, Cachorro); //Permite compilar. Muito perigoso!
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  Pessoa.Free;
  Cachorro.Free;
  op.Free;
end;
```

Embora o código da **Listagem 5** seja simples, há três coisas aqui que precisamos explicar. A primeira delas é que criamos os objetos TPessoa e TCachorro instanciando-os em variáveis do tipo TObject. Fizemos isso porque métodos que aceitam argumentos por referência, como o método Troca, só podem aceitar parâmetros do mesmo tipo que está declarado em seus argumentos, no caso TObject. Não poderíamos passar variáveis do tipo TPessoa ou TCachorro para esse método.

Outra coisa importante de notar é que de qualquer forma é impossível compilar o programa se você tentar usar o método Troca genérico. Não importa se as variáveis Pessoa e Cachorro são do tipo TObject ou se são do tipo TPessoa e TCachorro

respectivamente, você não conseguirá compilar o programa usando o método Troca nem com <TPessoa> e tampouco com <TCachorro>. A **Listagem 6** mostra mais alguns exemplos de código que não compilarão.

Listagem 6. Códigos que não compilarão por causa do tipo de dado

```
procedure TfrmGenerics.btEx2Click(Sender: TObject);
var
  Pessoa: TPessoa;
  Cachorro: TCachorro;
  op: TOperacao;
begin
  Pessoa  := TPessoa.Create('Vitor Rubio');
  Cachorro:= TCachorro.Create('Cujo');
  op := TOperacao.Create;
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  op.Troca<TPessoa>(Pessoa, Cachorro);
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  Pessoa.Free;
  Cachorro.Free;
  op.Free;
end;

procedure TfrmGenerics.btEx2Click(Sender: TObject);
var
  Pessoa: TPessoa;
  Cachorro: TCachorro;
  op: TOperacao;
begin
  Pessoa  := TPessoa.Create('Vitor Rubio');
  Cachorro:= TCachorro.Create('Cujo');
  op := TOperacao.Create;
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  op.Troca<TCachorro>(Pessoa, Cachorro);
  (Pessoa as TPessoa).DigaAlo;
  (Cachorro as TCachorro).Late;
  Pessoa.Free;
  Cachorro.Free;
  op.Free;
end;
```

A última observação que faremos a este código é que, trabalhando com TObject você é obrigado a fazer conversão de tipo para acessar os métodos das classes TPessoa e TCachorro. Usamos aqui a conversão segura, usando o operador "as".

Embora o exemplo tenha compilado normalmente o erro foi detectado somente ao tentar executar (Pessoa as TPessoa). DigaAlo; Como nós trocamos as variáveis e agora a variável Pessoa contém um objeto do tipo TCachorro o sistema dispara, corretamente, uma mensagem de "Invalid TypeCast" (Conversão de tipo inválida).

Muito mais perigoso seria não usar a conversão segura, mas usar a insegura, conforme podemos ver na **Listagem 7**.

Listagem 7. O perigosíssimo código com conversão de tipo insegura

```
procedure TfrmGenerics.btEx2Click(Sender: TObject);
var
  Pessoa: TObject;
  Cachorro: TObject;
  op: TOperacao;
begin
  Pessoa  := TPessoa.Create('Vitor Rubio');
  Cachorro:= TCachorro.Create('Cujo');
  op := TOperacao.Create;
  TPessoa(Pessoa).DigaAlo;
  TCachorro(Cachorro).Late;
  op.Troca(Pessoa, Cachorro);
  TPessoa(Pessoa).DigaAlo;
  TCachorro(Cachorro).Late;
  Pessoa.Free;
  Cachorro.Free;
  op.Free;
end;
```

O código da **Listagem** 7 é muito mais perigoso, porque podemos causar, no mínimo, um Access Violation, ou pior, misturar as coisas e fazer um cachorro falar e uma pessoa latir.

Classes Genéricas

Mostramos aqui como usar parâmetros genéricos em métodos, mas no nosso cotidiano será muito mais comum usar Generics em classes. Isso porque quando tivermos vários métodos e propriedades em uma classe e eles tiverem que ser flexíveis de acordo com o tipo de dado que vamos usar, devemos assegurar que todas as propriedades e métodos trabalhem com esse tipo de dado. A maioria das classes que usaremos com Generics serão classes que de alguma forma armazenam dados, como vetores, listas, filas e pilhas.

Primeiro vamos alterar nossa classe TOperacao para mostrar como será a sintaxe. Colocaremos também duas propriedades na classe TOperacao para armazenar temporariamente dois valores que serão trocados. Com isso os exemplos que fizemos até agora deixarão de funcionar, por isso alteraremos também os códigos dos eventos OnClick dos botões btEx1 e btEx2. A alteração que vamos fazer na classe TOperacao poderá ser vista na **Listagem 8**.

Listagem 8. Nova classe TOperacao

```
  TOperacao<TipoDeDado> = class
  private
    FEsquerda: TipoDeDado;
    FDireita: TipoDeDado;
  public
    property Esquerda: TipoDeDado read FEsquerda write FEsquerda;
    property Direita: TipoDeDado read FDireita write FDireita;
    procedure Troca;
  end;

implementation

{ TOperacao }
procedure TOperacao<TipoDeDado>.Troca;
var
  Temp: TipoDeDado;
begin
  Temp := FEsquerda;
  FEsquerda := FDireita;
  FDireita := Temp;
end;
```

Note que o parâmetro <TipoDeDado> manda em todos os lugares que existir uma variável do tipo TipoDeDado na classe inteira. Em todo lugar que usamos <TipoDeDado> este estará logo após o nome da classe e não após o nome do método.

O que fizemos foi criar dois campos privados, FEsquerda e FDireita, que serão acessados pelas variáveis Esquerda e Direita, todos esses do tipo TipoDeDado. O método Troca agora trabalha internamente com os campos da classe. Os valores trocados serão os valores das propriedades Esquerda e Direita.

Alteraremos nossos exemplos para continuar funcionando com essa nova classe. Basicamente vamos mover o <TipoDeDado> da execução do método para a criação do objeto, e armazenar os objetos instanciados nas propriedades Esquerda e Direita. O código das **Listagens 9** e **10** mostra como ficaram nossos exemplos tanto no botão btEx1 como no btEx2. O código do botão btEx2 deverá ser comentado pois não compilará. Ele

mostra que uma classe com um parâmetro genérico, uma vez instanciada, definindo-se o tipo, aceitará somente variáveis desse tipo.

Listagem 9. Os exemplos alterados

```
procedure TfrmGenerics.btEx1Click(Sender: TObject);
var
  Pessoa1, Pessoa2: TPessoa;
  opPessoa: TOperacao<TPessoa>;
  opNumeros: TOperacao<integer>;
  opDatas: TOperacao<TDateTime>;
begin
  Pessoa1 := TPessoa.Create;
  Pessoa2 := TPessoa.Create;
  opPessoa := TOperacao<TPessoa>.Create;
  opNumeros:= TOperacao<integer>.Create;
  opDatas:= TOperacao<TDateTime>.Create;
  Pessoa1.Nome := 'Blaise Pascal';
  Pessoa2.Nome := 'Albert Eninstein';
  opPessoa.Esquerda := Pessoa1;
  opPessoa.Direita := Pessoa2;
  opPessoa.Esquerda.DigaAlo;
  opPessoa.Direita.DigaAlo;
  ShowMessage('Trocando...');
  opPessoa.Troca;
  opPessoa.Esquerda.DigaAlo;
  opPessoa.Direita.DigaAlo;
  opNumeros.Esquerda := 1;
  opNumeros.Direita := 2;
  opNumeros.Troca;
  ShowMessage(IntToStr(opNumeros.Esquerda));
  ShowMessage(IntToStr(opNumeros.Direita));
  opDatas.Esquerda := StrToDate('14/02/1983');
  opDatas.Direita := Now;
  opDatas.Troca;
  ShowMessage(FormatDateTime('dd/mm/yyyy', opDatas.Esquerda));
  ShowMessage(FormatDateTime('dd/mm/yyyy', opDatas.Direita));
  Pessoa1.Free;
  Pessoa2.Free;
  opPessoa.Free;
  opDatas.Free;
  opNumeros.Free;
end;
```

Listagem 10. As propriedades que têm tipos pré-definidos não podem ser trocadas

```
procedure TfrmGenerics.btEx2Click(Sender: TObject);
var
  Pessoa: TPessoa;
  Cachorro: TCachorro;
  op: TOperacao<TPessoa>;
begin
{
  Pessoa  := TPessoa.Create('Vitor Rubio');
  Cachorro:= TCachorro.Create('Cujo');
  op := TOperacao<TPessoa>.Create;
  op.Esquerda := Pessoa;
  op.Direita := Cachorro;
  op.Troca;
  op.Esquerda.DigaAlo;
  op.Direita.DigaAlo;
  Pessoa.Free;
  Cachorro.Free;
  op.Free;
}
end;
```

O mais interessante da **Listagem 9** é que como definimos opPessoa como opPessoa: TOperacao<TPessoa> e criamos como opPessoa := TOperacao<TPessoa>.Create podemos contar em tempo de Design com os recursos de Autocomplete do IDE e mais qualquer outro recurso em tempo de Design do Delphi. Para todos os efeitos, as variáveis Esquerda e Direita de opPessoa são do tipo TPessoa, e as variáveis Esquerda e Direita de opNumeros são do tipo Integer. Usando Generics pudemos transformar uma classe em três diferentes, uma para trabalhar com datas, outra para números inteiros e ainda outra para objetos do tipo TPessoa.

O código da **Listagem 10** simplesmente não compila. Não há meios, nem o risco, de se atribuir um objeto do tipo cachorro à propriedade Direita ou Esquerda. TOperacao<TCachorro> poderia trabalhar somente com cachorros, ou TOperacao<TObject> trabalhar com objetos, mas este último não faz sentido, pois exigiria uma conversão de tipo desnecessária, a não ser que queiramos propositalmente misturar os tipos de objetos.

Um exemplo mais prático

No exemplo que criaremos agora vamos criar uma classe TPilha, que implementa a estrutura de dados pilha. A nossa classe será genérica, ou seja, poderá trabalhar com qualquer tipo de dados. Não será necessário criar um descendente de TPilha para trabalhar com cada tipo de dado, e não serão necessárias conversões de tipo para adicionar ou remover elementos da pilha. Crie uma nova unit no nosso programa chamada uPilhas. Vamos criar nessa unit a classe TPilha conforme a **Listagem 11**.

Nota

Quando criar um tipo genérico que possui internamente um vetor, use um vetor dinâmico, que funciona com qualquer tipo de dado. Se tentar usar um vetor estático de tamanho fixo, a classe genérica funcionará bem quando o tipo de dado passado for, por exemplo, Integer ou ShortString, mas se você tentar usar String, devido a um bug, ocorrerá uma exceção "Access Violation" na hora de destruir o objeto. A **Listagem 20** no final do artigo mostra isso.

A **Listagem 11** é uma implementação básica de pilhas, retirada de exercícios comuns da disciplina de Estrutura de Dados, lecionada em cursos e faculdades tecnológicos. Adaptei o código para o contexto da nossa aplicação, que é Delphi 2010, orientado a objetos, com Generics.

O método Push adiciona um elemento, o método Pop retira o elemento que estiver no topo e o método Top apenas obtém o elemento que estiver no topo, porém, sem extraí-lo. O máximo de elementos permitidos nesta pilha é 255. O vetor onde são armazenados os elementos é um array dinâmico. No construtor da classe, setamos o tamanho do vetor para 255 com a procedure SetLegth, e no destructor setamos o tamanho para 0. Caso tente extrair um elemento depois que a pilha estiver vazia ou acrescentar um elemento depois que a pilha atingir 255 elementos, será disparada uma exceção. Embora limitemos a nossa pilha a 255 elementos, a classe TStack da biblioteca do Delphi

Nota do DevMan

Pilha é uma estrutura de dados em que são inseridos da mesma forma que uma pilha de pratos, de baixo para cima. Assim como uma pilha de pratos, ou um porta guardanapo de lanchonete, o último elemento inserido numa pilha é o primeiro a sair e o primeiro elemento é o último a sair. Chamamos esse comportamento de LIFO, acrônimo para "Last In First Out" ou "Último a entrar primeiro a sair". Embora implementaremos uma classe básica para se trabalhar com pilhas, o Delphi já possui, na unit Generics.Collections a classe TStack<T> que atende essa necessidade e será vista mais a frente. Não abordaremos o funcionamento teórico da estrutura de dados pilha porque foge do escopo desse artigo.

Listagem 11. Implementação de TPilha

```
unit uPilhas;

interface

uses
  SysUtils;

const MAX = 255;

type
  TPilha<TipoDeDado> = class
  private
    FMemo : array of TipoDeDado;
    FTopo : Integer;
  public
    constructor Create;
    destructor Destroy; override;
    procedure Clear;
    function IsEmpty: boolean;  {verifica se vazia}
    function IsFull: boolean;    {verifica se cheia}
    procedure Push(const  x:TipoDeDado); {empilha}
    function Pop:TipoDeDado;             {desempilha}
    function Top:TipoDeDado;             {retorna o elemento do
     topo sem desempilhar}
    property Topo: Integer read FTopo write FTopo; {retorna o
     índice do topo}
  end;

implementation

{ TPilha<TipoDeDado> }

procedure TPilha<TipoDeDado>.Clear;
begin
  while not IsEmpty do
    Pop;
  SetLength(FMemo, 0);
end;

constructor TPilha<TipoDeDado>.Create;
begin
  inherited Create;
  SetLength(FMemo, MAX);
  FTopo := 0;
end;

destructor TPilha<TipoDeDado>.Destroy;
begin
  Clear;
  inherited;
end;

function TPilha<TipoDeDado>.IsEmpty: boolean;
begin
  if FTopo=0 then
    Result := true
  else
    Result := false;
end;

function TPilha<TipoDeDado>.IsFull: boolean;
begin
  if FTopo = MAX then
    Result := true
  else
    Result := false;
end;

function TPilha<TipoDeDado>.Pop: TipoDeDado;
begin
  if not IsEmpty then
  begin
    Result := FMemo[FTopo];
    Dec(FTopo);
  end
  else
    raise Exception.Create('Stack Underflow');
end;

procedure TPilha<TipoDeDado>.Push(const x: TipoDeDado);
begin
  if not IsFull then
  begin
    inc(FTopo);
    FMemo[FTopo] := x;
  end
  else
    raise Exception.Create('Stack Overflow!');
end;

function TPilha<TipoDeDado>.Top: TipoDeDado;
begin
  if not IsEmpty then
    Result := FMemo[FTopo]
  else
    raise Exception.Create('Stack Underflow');
end;

end.
```

não possui essa limitação. Seu vetor dinâmico de elementos cresce automaticamente. Na classe TStack não existe o método Top, mas existe o método Peek que faz a mesma coisa.

Adicione mais três botões ao form, chamados btEx3, btEx4 e btEx5 respectivamente, e use o código da **Listagem 12** para testarmos as pilhas. Coloque uPilhas no uses de sua unit principal. A **Listagem 12** mostra como podemos usar nossa classe para criar pilhas de qualquer tipo de dado.

Como vemos, nas pilhas o último elemento que inserimos é o primeiro a sair. Os elementos sempre saem em ordem inversa. Um destaque para o código do evento OnClick do botão btEx5, que usa a classe TStack do Delphi. Veremos mais adiante que o Delphi possui outras classes prontas que já fazem uso de Generics.

Constraints nos tipos genéricos

Nos Generics constraints servem para limitar ou pré-definir que tipo de dado será usado no parâmetro genérico. Sua sintaxe é formada por dois pontos ":" após o identificador do parâmetro genérico seguidos de uma palavra – chave. Essa palavra chave pode ser o nome de uma classe, de uma interface, a palavra "class" ou a palavra "constructor", podendo ter várias dessas constraints separadas por vírgulas.

Quando tivermos <TipoDeDado: class> o parâmetro TipoDeDado deverá ser uma classe qualquer. Não se deve usar <TipoDeDado: TObject>. Quando tivermos <TipoDeDado: TPessoa> somente serão aceitos a classe TPessoa ou descendentes. Para aceitar qualquer tipo de dado que implemente uma interface devemos fazer <TipoDeDado: [InterfaceQueDesejamos]>. Se precisarmos criar um objeto qualquer, do tipo TipoDeDado, dentro da nossa classe genérica precisaremos acessar o constructor desse tipo. Para isso devemos usar a palavra chave constructor, assim: <TipoDeDado: constructor>. Os próximos exemplos nos mostrarão como usar constraints e como criar um Factory Method bem facilmente.

Factory Method, ou método fábrica, é um padrão de projeto bastante conhecido. Usamos este padrão para isolar units e diminuir o acoplamento entre os módulos do nosso sistema.

Por exemplo, quando um objeto A precisa de um objeto B, mas não sabe quem é o objeto B, não sabe em que unit ele está implementado, não sabe como criá-lo ou como configurá-lo e não precisa saber disso nós usamos um terceiro método, de uma outra classe, que tem esse conhecimento e sabe como criar o objeto B. Então o método fábrica cria o objeto B e o entrega-o para o objeto A. Embora a unit da classe A não tenha a unit da classe B declarada em seu uses, e nem

o contrário, a unit da classe A possui a unit do método fábrica em seu uses, e esta possui a unit da classe B. Usando Generics podemos criar métodos fábrica que possam criar qualquer tipo de objeto, mesmo que não tenha acesso à unit onde esses estão declarados.

Listagem 12. Exemplo de uso de pilhas

```
procedure TfrmGenerics.btEx3Click(Sender: TObject);
var
  PilhaInteiro: TPilha<integer>;
begin
  PilhaInteiro := TPilha<integer>.Create;
  PilhaInteiro.Push(1);
  PilhaInteiro.Push(2);
  PilhaInteiro.Push(3);
  PilhaInteiro.Push(4);
  ShowMessage(IntToStr(PilhaInteiro.Top));
  ShowMessage(IntToStr(PilhaInteiro.Pop));
  ShowMessage(IntToStr(PilhaInteiro.Pop));
  ShowMessage(IntToStr(PilhaInteiro.Pop));
  ShowMessage(IntToStr(PilhaInteiro.Pop));
  PilhaInteiro.Free;
end;

procedure TfrmGenerics.btEx4Click(Sender: TObject);
var
  PilhaString: TPilha<string>;
begin
  PilhaString := TPilha<string>.Create;
  PilhaString.Push('primeiro');
  PilhaString.Push('segundo');
  PilhaString.Push('ultimo');
  ShowMessage(PilhaString.Top);
  ShowMessage(PilhaString.Pop);
  ShowMessage(PilhaString.Pop);
  ShowMessage(PilhaString.Pop);
  PilhaString.Free;
end;

procedure TfrmGenerics.btEx5Click(Sender: TObject);
var
  PilhaString: TStack<string>;
begin
  PilhaString := TStack<string>.Create;
  PilhaString.Push('primeiro');
  PilhaString.Push('segundo');
  PilhaString.Push('ultimo');
  ShowMessage(PilhaString.Peek);
  ShowMessage(PilhaString.Pop);
  ShowMessage(PilhaString.Pop);
  ShowMessage(PilhaString.Pop);
  PilhaString.Free;
end;
```

Listagem 13. Unit uPessoa

```
unit uPessoa;

interface

uses
  Dialogs;

type
  TPessoa = class
  private
    FNome: string;
  public
    procedure DigaAlo;
    constructor Create(umNome: string);  overload;
    property Nome: string read FNome write FNome;
  end;

implementation

{ TPessoa }

constructor TPessoa.Create(umNome: string);
begin
  FNome := umNome;
end;

procedure TPessoa.DigaAlo;
begin
  ShowMessage('Olá, meu nome é ' + FNome);
end;

end.
```

Vamos fazer uma grande alteração no nosso exemplo. Vamos separar cada classe que criamos em uma unit diferente, então teremos uma unit uPessoa para a classe TPessoa, uma unit uOperacao para a classe TOperacao e uma unit uCachorro para a classe TCachorro. Faremos algumas alterações nessas classes e também nos códigos dos botões btEx1 e btEx2. Coloque as novas units no uses da unit principal. A **Listagem 13** contém o código da unit uPessoa.

Colocamos no uses da unit uPessoa a referência à Dialogs para termos acesso à função ShowMessage, porém a classe TPessoa permanece a mesma. Não fizemos nenhuma outra alteração que mereça consideração. A **Listagem 14** possui o código da unit uCachorro, análoga a unit uPessoa.

Listagem 14. Unit uCachorro

```
unit uCachorro;

interface

uses
  Dialogs;

type
  TCachorro = class
  private
    FNome: string;
  public
    procedure Late;
    constructor Create(umNome: string); overload;
    property Nome: string read FNome write FNome;
  end;

implementation

{ TCachorro }

constructor TCachorro.Create(umNome: string);
begin
  FNome := umNome;
end;

procedure TCachorro.Late;
begin
  ShowMessage('rrrrrr au, au! ('+FNome+')');
end;

end.
```

Apenas na linha ShowMessage('rrrrrr au, au! ('+FNome+')'); fizemos uma alteração para saber qual cachorro está latindo. A unit uOperacao podemos ver na **Listagem 15**. Nenhuma mudança significativa nessa unit.

A unit uFabrica contém a classe genérica TFabrica<TipoDeDado>. Esta classe tem um método de classe chamado CriaOperacao. Esse método é responsável por criar o objeto do tipo TOperacao<TipoDeDado>, sendo este TipoDeDado o mesmo passado à classe TFabrica. A fábrica criará não só o objeto TOperacao mas também criará os dois operandos, Esquerda e Direita, todos do mesmo tipo: TipoDeDado. A **Listagem 16** mostra como deve ficar o código.

A grande novidade aqui é a constraint "constructor". Essa constraint permite que sejam criados objetos do tipo TipoDeDado dentro do contexto da TOperacao. Sem essa constraint não seria possível executar o constructor create. Essa classe fábrica possibilita que seja entregue à unit principal um objeto do tipo TOperacao já montado, com dois objetos TipoDeDado criados e associados às suas propriedades Esquerda e Direita, podendo estas ser do tipo TPessoa ou descendentes, ou TCachorro ou descendentes. Criamos uma boa dose de reaproveitamento e economia de código!

Listagem 15. Unit uOperacao

```
unit uOperacao;

interface

type

  TOperacao<TipoDeDado> = class
  private
    FEsquerda: TipoDeDado;
    FDireita: TipoDeDado;
  public
    property Esquerda: TipoDeDado read FEsquerda write FEsquerda;
    property Direita: TipoDeDado read FDireita write FDireita;
    procedure Troca;
  end;

implementation

{ TOperacao }
procedure TOperacao<TipoDeDado>.Troca;
var
  Temp: TipoDeDado;
begin
  Temp := FEsquerda;
  FEsquerda := FDireita;
  FDireita := Temp;
end;

end.
```

Listagem 16. O método fábrica

```
unit uFabrica;

interface

uses
  uOperacao;

type
  TFabrica<TipoDeDado: constructor> = class
  public
    class function CriaOperacao: TOperacao<TipoDeDado>;
  end;

implementation

{ TFabrica<TipoDeDado> }

class function TFabrica<TipoDeDado>.CriaOperacao: TOperacao<TipoDeDado>;
begin
  Result := TOperacao<TipoDeDado>.Create;
  Result.Esquerda := TipoDeDado.Create;
  Result.Direita := TipoDeDado.Create;
end;

end.
```

A **Listagem 17** mostra o código dos botões btEx1 e btEx2, como ficaram simples. A classe TFabrica é genérica e ela cria uma TOperacao também genérica. É importante lembrar que como nossa fábrica tem a constraint constructor, não podemos usar com ela tipos de dados primitivos, como integer, ou tipos estruturados como records.

É importante observar na **Listagem 17** que, embora não tenhamos que nos preocupar com a criação da TOperacao e suas partes, temos que nos preocupar com sua destruição. Para isso chamamos o método Free da Esquerda e da Direita, antes de executar o Free da própria operação.

Genéricos prontos de fábrica

O Delphi 2010 já vem com vários tipos genéricos prontos para usar. Já vimos a classe TStack<T>, para se trabalhar com pilhas. Temos também as classes TQueue<T>, para se trabalhar com filas, TList<T> para trabalhar com listas e mais três classes análogas a estas, TObjectStack, TObjectQueue e TObjectList para se trabalhar especificamente com objetos.

Listagem 17. Botões btEx1 e btEx2 alterados

```
procedure TfrmGenerics.btEx1Click(Sender: TObject);
begin
  with TFabrica<TPessoa>.CriaOperacao do
  begin
    Esquerda.Nome := 'Blaise Pascal';
    Direita.Nome :=  'Albert Eninstein';
    Esquerda.DigaAlo;
    Direita.DigaAlo;
    ShowMessage('Trocando...');
    Troca;
    Esquerda.DigaAlo;
    Direita.DigaAlo;
    Esquerda.Free;
    Direita.Free;
    Free;
  end;end;

procedure TfrmGenerics.btEx2Click(Sender: TObject);
begin
  with TFabrica<TCachorro>.CriaOperacao do
  begin
    Esquerda.Nome := 'Cujo';
    Direita.Nome := 'Marley';
    Esquerda.Late;
    Direita.Late;
    ShowMessage('Trocando...');
    Troca;
    Esquerda.Late;
    Direita.Late;
    Esquerda.Free;
    Direita.Free;
    Free;
  end;
end;
```

Na unit Generics.Collections.pas temos ainda a classe abstrata TEnumerable, cujos descendentes podem ser coleções navegáveis com o operador for...in. Temos também o tipo TDictionary<TKey, TValue> que trabalha com dois tipos genéricos. Essa classe pode gerar uma Hash Table, que é uma lista de pares Chave-Valor.

Elaboraremos agora alguns exemplos do tipo "Antes – Depois" que mostrará como fazíamos algumas coisas sem os genéricos antigamente e como podemos fazer agora. Vamos adicionar mais dois botões, chamados btAntes e btDepois com os Captions "Antes" e "Depois" respectivamente. Nosso formulário ficará como a **Figura 1**.

No evento OnClick do botão btAntes faremos o código da **Listagem 18**. Simplesmente criaremos uma lista de valores e tentaremos navegar por ela, ilustrando a dificuldade de se fazer isso sem Generics.

Coloque no uses a unit Contnrs, que contém as coleções "antigas" do Delphi.

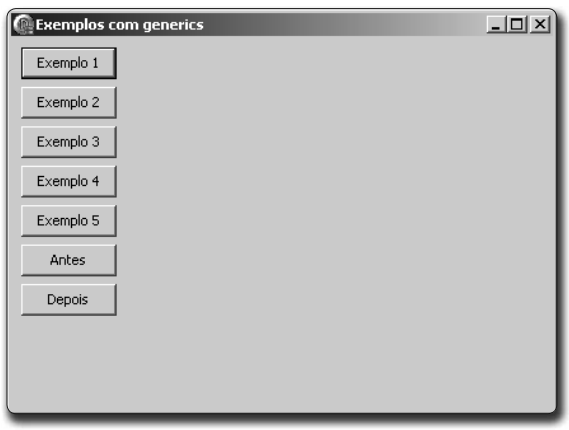


Figura 1. O formulário principal, com todos os botões

Listagem 18. O Antes

```
procedure TfrmGenerics.btAntesClick(Sender: TObject);
var
  Lista: TList;
  ListaObj: TObjectList;
  i1, i2: integer;
  p1, p2: TPessoa;
  i, j: Integer;
begin
  Lista := TList.Create;
  ListaObj := TObjectList.Create;
  i1 := 1;
  i2 := 2;
  p1 := TPessoa.Create('José');
  p2 := TPessoa.Create('Manoel');
  Lista.Add(@i1);
  Lista.Add(@i2);
  ListaObj.Add(p1);
  ListaObj.Add(p2);
  //varrendo lista
  for i :=  0 to Lista.Count - 1 do
    ShowMessage(IntToStr(integer(Lista[i]^)));
  //varrendo listaObj
  for j :=  0 to ListaObj.Count - 1 do
    ShowMessage((ListaObj[j] as TPessoa).Nome);
  Lista.Free;
  ListaObj.Free;
end;
```

Antigamente, para usar TList, éramos obrigados a usar ponteiros, qualquer que fosse o tipo a armazenar. Exceto para armazenar objetos, que já são ponteiros. Para poder extrair um ponteiro de um Integer a fim de armazenar na lista, tínhamos que usar o operador de endereço @. Analogamente, para se trabalhar com TObjectList era necessário fazer conversão de tipo na saída, sempre. Mesmo assim ao adicionar itens na lista nenhuma verificação é feita. Na **Listagem 19** podemos ver o uso de listas genéricas, tanto para objetos como para tipos primitivos.

Listagem 19. O Depois

```
procedure TfrmGenerics.btDepoisClick(Sender: TObject);
var
  Lista: TList<integer>;
  ListaObj: TObjectList<Tpessoa>;
  i: integer;
  p: TPessoa;
begin
  Lista := TList<integer>.Create;
  ListaObj := TObjectList<Tpessoa>.Create;
  Lista.Add(1);
  Lista.Add(2);
  ListaObj.Add(TPessoa.Create('José'));
  ListaObj.Add(TPessoa.Create('Manoel'));
  //varrendo lista
  for i in Lista do
    ShowMessage(IntToStr(i));
  //varrendo listaObj
  for p in ListaObj do
    ShowMessage(p.Nome);

  Lista.Free;
  ListaObj.Free;
end;
```

O primeiro dos benefícios é a clareza do código. Podemos adicionar um inteiro à lista direto, simplesmente com Add(1). Analogamente podemos adicionar um objeto criando diretamente. Não é mais necessário criar descendentes de TObjectList para se controlar o que entra ou sai, como fazíamos antigamente. E o operador for...in é um grande avanço (corresponde ao for each do C# e do Delphi Prism).

Um alerta importante é atenção ao se criar tipos genéricos contendo arrays internos. Arrays estáticos causam um bug ao se usar Strings. Mas não há necessidade de se preocupar com as coleções genéricas que já vêm prontas no Delphi, pois todas elas usam vetores dinâmicos. A **Listagem 20** apresenta um bug que ocorre quando o tipo genérico possui internamente um array estático. O mesmo não ocorre se o array for dinâmico.

Listagem 20. Erro com array estático

```
  TTeste<T> = class
  private
    FTeste: array [1..10] of T;
  end;

{...}

procedure TfrmGenerics.btErroClick(Sender: TObject);
var
  teste1: TTeste<integer>;
  teste2: TTeste<shortstring>;
  teste3: TTeste<string>;
begin
  teste1 := TTeste<integer>.Create;
  teste2 := TTeste<shortstring>.Create;
  teste3 := TTeste<string>.Create;
  teste1.FTeste[1] := 0;
  teste2.FTeste[1] := 'teste2';
  teste3.FTeste[1] := 'teste3';
  ShowMessage('liberando teste1');
  teste1.Free;
  ShowMessage('liberando teste2');
  teste2.Free;
  //as linhas abaixo causarão um erro
  //ShowMessage('liberando teste3');
  //teste3.Free;
end;
```

Conclusão

Generics são uma nova ferramenta a favor da nossa produtividade. Além de ser um recurso moderno e elegante, os tipos genéricos nos permitem economizar bastante do tempo que normalmente gastaríamos, criando várias classes ou vários métodos overload para se trabalhar com tipos diferentes. Usando essa nova ferramenta nos livramos de várias conversões de tipo enfadonhas que tendem a poluir o código.

O Delphi possui várias classes já prontas que usam genéricos, portanto vamos usá-las. Essas ferramentas conferem ao Delphi para win32 a mesma flexibilidade e produtividade que antigamente só encontraríamos no .Net.

Embora não tenhamos explorado todo o potencial dos tipos parametrizados, já fornecemos um vislumbre do que será possível fazer daqui em diante. Por exemplo, componentes que se ligam a outros componentes genéricos. Editores de coleções ou contêineres de itens, como os campos de um DataSet genérico. Eventos genéricos. As possibilidades são enormes, basta usar a criatividade ●

Links

Blogs do autor - http://twitter.com/vitorrubio

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

MVC com PHP na prática

Migre passo a passo uma aplicação OO para usar o poderoso padrão MVC

Lucas Simões Maistro
lucassimoes@univem.edu.br
Analista/Programador do Centro Universitário Eurípides de Marília. É desenvolvedor Delphi desde 1999 e trabalha com Oracle e SQL Server desde 2001. Graduado em Análise de Sistemas e Pós Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Raphael Zanon Rodrigues
zanonr@gmail.com
É Analista/Programador do Centro Universitário Eurípides de Marília. Desenvolvedor Delphi. Graduado em Análise de Sistemas e Pós Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Resumo do DevMan

As técnicas empregadas no desenvolvimento de software são essenciais para o sucesso do mesmo. Boas práticas como TDD, padrões de projetos, orientação a objetos e MVC estão em constante destaque na internet e publicações especializadas. Com isso, uma técnica que vem tendo bastante destaque é o MVC, porém poucas pessoas sabem como migrar a arquitetura de suas aplicações OO para a arquitetura MVC. Neste artigo primeiramente iremos tratar de toda parte conceitual que envolve o MVC, e em seguida construiremos um pequeno projeto OO onde utilizaremos as funções básicas de Insert, Update, Delete e Select para na sequência modificarmos passo a passo a arquitetura do projeto para MVC.

Do que trata o artigo
Este artigo mostra como fazer com que uma aplicação PHP orientada a objetos utilize a arquitetura MVC através de um exemplo simples, porém que utilize os recursos básicos de persistência e consulta a base de dados MySQL.

Para que serve
A orientação a objetos é a melhor forma para se construir uma aplicação, porém muito se fala na arquitetura MVC, que divide em camadas as classes existentes e, entre outras coisas, facilita o desenvolvimento em equipe.

Em que situação o tema é útil
A arquitetura MVC organiza o código e facilita o desenvolvimento em equipes multidisciplinares. Com o exemplo aqui apresentado será possível entender como tornar uma aplicação orientada a objetos em uma aplicação MVC.

Visando o crescente aumento da necessidade de desenvolvimento de sistemas com prazo quase sempre curtos, seja por falha no levantamento de informações, definição de requisitos, projeto ou qualquer outro fator, nesse artigo, visamos abordar um assunto bastante pertinente a todos nós desenvolvedores com foco em um projeto de sistema para internet. Iremos abordar o padrão de arquitetura da aplicação, o padrão MVC.

Resumidamente, o padrão MVC, que significa Model-View-Controller, tem

como objetivo a divisão de tarefas específicas em um sistema, aumentando a produtividade de desenvolvimento do projeto, pois facilita a identificação e alteração em qualquer parte dele, seja em um módulo ou no visual.

Padrões de Projeto (Design Patterns)

Padrões de Projetos podem, basicamente, serem vistos como uma solução já testada para problemas que alguém já passou e o solucionou aplicando um modelo básico que foi documentado, permitindo que nós utilizemos esse modelo integralmente ou adaptado de acordo com as necessidades da nossa solução, o que resulta na redução do tempo gasto com o desenvolvimento e possíveis melhorias na qualidade de uma aplicação.

Um Padrão de Projeto tem como características a Generalidade, Equilíbrio, Abstração e Combinatoriedade, ou seja, todo padrão deve permitir a construção de outros projetos a partir dele, permitir o relacionamento de cada uma das restrições encontradas com cada passo do projeto, possibilitar a aplicação de soluções adquiridas por meio de abstrações feitas sobre conhecimentos adquiridos em outras experiências e o relacionamento de forma hierárquica com outros padrões.

A utilização de Padrões concede, de forma geral, vantagens como redução no tempo de codificação, reaproveitamento de código, diminuição no tempo de manutenção, aumento na interação entre a equipe de desenvolvimento, padronização do código e a possibilidade de generalização do sistema. O grande desafio das equipes de desenvolvimento de aplicações é produzir aplicativos cada vez mais seguros, eficientes, de fácil manutenção, reutilizáveis e em prazos cada vez menores.

Quando e como utilizar um padrão para desenvolvimento de uma solução? Atualmente, o principal desafio para as equipes responsáveis pelo desenvolvimento de soluções é criar aplicativos seguros, eficientes, com manutenção simples, códigos reutilizáveis e ainda, em prazos cada vez menores. Dessa forma, a primeira coisa que devemos ter em mente é o bom senso, ou seja, realizar o levantamento dos requisitos e definir o escopo do projeto. Depois dessa primeira fase devemos identificar possíveis deficiências e se o projeto pode ser otimizado. Feito isso, se for o caso, utilizaremos o padrão de projeto que se ajusta melhor à solução para sanar as deficiências levantadas.

Nos últimos tempos, o paradigma do modelo orientado a objetos vem tendo avanço significante, o que torna impensável o fato de se desenvolver novos projetos que não sejam orientados a objetos desde o início. Porém, para atingirmos com sucesso o desenvolvimento desses novos projetos, torna-se cada vez mais necessário, levarmos em consideração a arquitetura que vamos utilizar para tal. Com base nas tendências e observação dos padrões utilizados no mercado, chegamos à conclusão de que a organização desses novos projetos será em camadas.

O desenvolvimento de novos projetos com o conceito de organização em camadas é, basicamente, utilizar cada tecnologia para a finalidade em que ela foi desenvolvida. Organizando nossos projetos em camadas atingimos a independência das mesmas e a partir de então é que conseguiremos atingir os objetivos de desenvolver aplicações seguras, eficientes, com manutenção simples e códigos reutilizáveis.

Desenvolvimento em Camadas

A nomenclatura "camadas" pode parecer um tanto quanto nova, porém, desde o surgimento da arquitetura cliente/servidor, o desenvolvimento de aplicações em camadas passou a ser adotado, mas com a popularização da internet, os cuidados com a arquitetura das aplicações têm ganhado cada vez mais visibilidade. Mesmo assim, entendemos que o desenvolvimento de aplicações multicamadas pode parecer em um primeiro instante, mais complexo do que realmente é. Pensando nessa dificuldade, vamos explicar como foi a evolução desse conceito.

Em um primeiro momento, as aplicações eram criadas para serem usadas em uma única máquina. Dessa forma, tais aplicações continham apenas um único módulo com todas as funcionalidades como: validações, lógica de negócio e acesso a dados, incluídas nele o que resultava em uma enorme quantidade de linhas e bastante sofrimento quando tinha necessidade de manutenção. A **Figura 1** mostra como era esse tipo de aplicação.

Figura 1. Modelo em uma única camada

Passado algum tempo, onde os computadores começaram a se popularizar, surge a necessidade de se ter vários usuários acessando de forma simultânea a base de dados. Nesse ponto, a base de dados foi colocada em uma máquina separada das máquinas que utilizavam as aplicações. Essa arquitetura trouxe vantagens em relação a sua antecessora, aumentando a velocidade na rede e o processamento deixou de ser totalmente centralizado. Temos na primeira camada o cliente, onde se encontra a apresentação, as regras de negócio e o acesso aos dados, que se conectam a uma segunda camada (servidor), onde está o banco de dados (**Figura 2**).

As aplicações três camadas, também conhecidas como multicamadas ou *multitier*, são uma evolução do modelo cliente / servidor, onde surge uma (ou mais) camada intermediária entre a parte cliente e o servidor de dados. O ponto fundamental desse modelo é a divisão distinta entre as camadas onde não há a intromissão de uma na outra. A divisão é feita entre as camadas de apresentação, de negócio e de dados, como mostra a **Figura 3**.

Arquitetura MVC: Controlador, Modelo e Visão

O padrão MVC tem como princípio que o desenvolvedor tem três peças que trabalham em conjunto para formar uma aplicação

mais complexa, sendo criada originalmente para resolver o problema de interação do usuário com a aplicação, deixando que essa interação propriamente dita ficasse isolada das regras de negócio. Para exemplificar vamos utilizar a ideia de um carro. O carro possui duas Visões (Views), sendo elas o exterior e o interior. Ambas pegam as ações de um Controlador (Controller), o motorista. Os freios, aceleração, partida e outras funções representem o Modelo (Model). Como funciona? As Visões capturam os eventos externos, o Controlador (motorista) interpreta os eventos "crus" dados pela visão passando para o Modelo quando necessário, que por sua vez devolve os resultados às Visões.

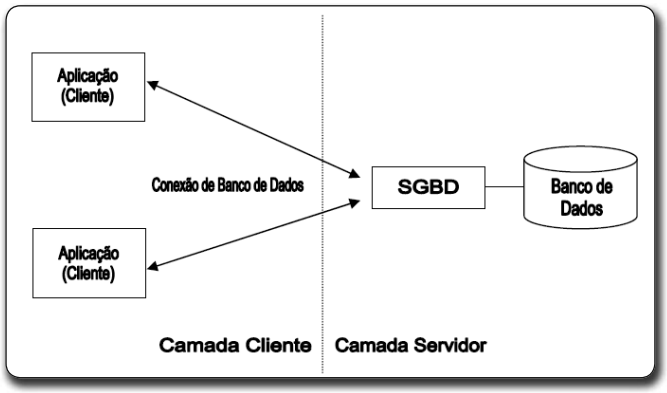


Figura 2. Modelo em duas camadas

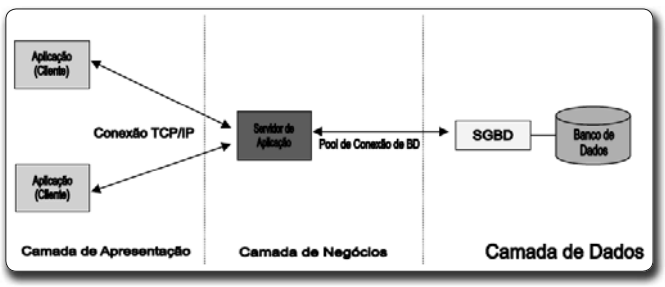


Figura 3. Modelo em três camadas

Com essa divisão, o padrão de arquitetura MVC atinge o objetivo de separar as funcionalidades envolvidas na manutenção e apresentação dos dados de uma aplicação. A **Figura 4** demonstra a aplicação do padrão MVC com a preocupação em separar a informação da apresentação.

MVC na Web

Em um primeiro momento, sistemas web eram normalmente um aglomerado de páginas dentro de um mesmo contexto, onde cada página geralmente continha todo o código necessário para seu funcionamento, desde a apresentação até a comunicação com os dados, de uma forma praticamente procedural. Com o tempo, alternativas para a esta forma de utilização foram introduzindo novos padrões num conjunto de melhores práticas, como o de Orientação a Objetos, que começa a separar, mesmo que dentro de apenas um arquivo, funcionalidades da página.

A ideia por trás da implementação de uma aplicação com o modelo MVC é a extensibilidade, deixando a aplicação bastante simples e extremamente flexível, tendo normalmente o objetivo de criar um controlador bastante eficiente e os mecanismos de comunicação entre as partes. Por este padrão separar tão bem as camadas, ele vem sendo cada vez mais utilizado para o desenvolvimento de aplicações Web, pois com base em cada ação do usuário e no resultado do processamento do modelo, fica fácil para o controlador selecionar qual visão será exibida como resposta à solicitação do usuário.

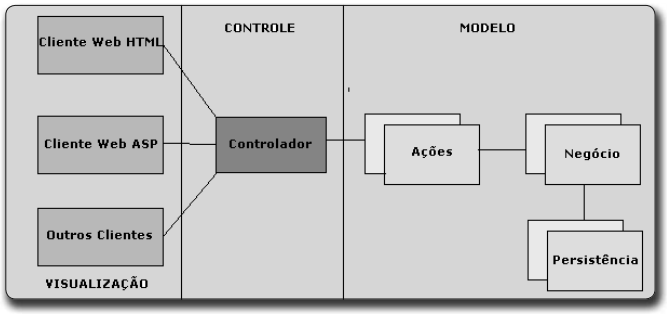


Figura 4. Modelo no padrão MVC

É bastante comum que uma página HTML tenha a capacidade de processamento utilizando tecnologias que atuam do lado do cliente, como por exemplo, o JavaScript que é escrito pelo componente Visão. Nesse ponto, é comum que as pessoas pensem que o componente Visão é tudo o que acontece no navegador, porém, a interação que acontece no navegador com o usuário também segue o padrão MVC. A camada Visão que fica no servidor atua, de certa forma, como um modificador da implementação do padrão MVC no navegador. Utilizando a tecnologia JavaScript torna-se possível atuar com o papel de Controle, modificar o visual e inclusive os dados que o usuário vê. Temos então, um exemplo bastante simples de como o padrão MVC pode ser aplicado a apenas uma camada de cada vez, onde a camada de um nível mais abaixo vai ter que se comunicar com o modelo referente à camada de um nível superior, ou, se comunicar com a aplicação como um todo, mas, a camada superior não pode se comunicar com nada além da visão referente à camada inferior.

Decorrente do fato do navegador ter uma função fundamental na interação da aplicação com o usuário, a implementação MVC como explicada no parágrafo anterior deve ser entendida como uma camada essencial para a aplicação, ou seja, essa implementação nos leva a criar uma camada em JavaScript que roda no navegador se comunicando com o servidor de forma assíncrona, o que acabou gerando o padrão AJAX.

A utilização do modelo MVC para o desenvolvimento de aplicações tem algumas vantagens como gerenciar vários visualizadores com o mesmo modelo, fácil para manter, testar e atualizar múltiplos sistemas; facilidade para inclusão de novos clientes, apenas gerando suas visões e controles; torna a aplicação escalável; e a possibilidade do desenvolvimento em paralelo para as camadas devido ao fato de serem independentes; e desvantagens como a necessidade da utilização de um maior tempo para analisar e modelar a aplicação; necessidade de pessoal especializado para o desenvolvimento, que tornam

mais complicado o desenvolvimento de aplicações de pequeno porte, entre outras que devem ser levadas em consideração.

Considerações iniciais do projeto

Este artigo tem a intenção de mostrar ao leitor como separar as classes de seus projetos dentro das camadas MVC. Sendo assim, teremos no primeiro exemplo um arquivo index.php que irá realizar as solicitações aos métodos do arquivo aluno.class.php, que é a classe Aluno. Em seguida, iremos começar um novo projeto, e este será composto pelo mesmo arquivo index.php com alterações mínimas, e pelas classes no formato MVC, ou seja, os arquivos aluno_controller.class.php, aluno_view.class.php e aluno_model.class.php. Veja na **Figura 5** o diagrama dos dois projetos.

Nota

Para o nosso exemplo, utilizaremos o WampServer, que é um pacote para o Windows que efetua a instalação automática das ferramentas Apache, MySQL e PHP. Quando instalado, o Wamp ativa um servidor Apache e um servidor MySQL, criando um ambiente de testes, permitindo aos desenvolvedores web criarem aplicações utilizando as ferramentas instaladas.

Para instalar o WampServer, basta efetuar o download da versão mais atual na sessão de downloads no site do desenvolvedor (veja a sessão links), seguir com a instalação, que não requer nenhum conhecimento adicional, e logo após o término da instalação os scripts PHP já estarão disponíveis.

Vale lembrar que o Wamp utiliza a porta 80. A questão é que o IIS (Internet Information Services) e outros programas de acesso remoto (ex. TeamViewer) também usam essa porta, sendo assim é necessário manter esses serviços parados enquanto efetuamos nossos testes utilizando o Wamp.

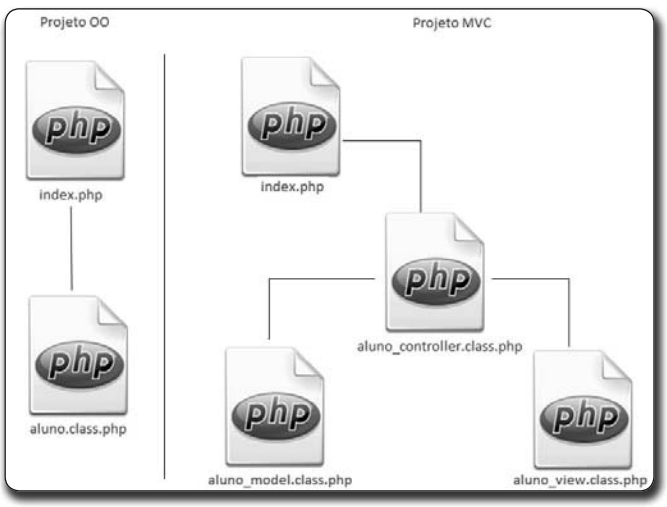


Figura 5. Projetos OO e MVC

Criando o banco de dados

O banco de dados será composto por uma única tabela onde serão armazenados os alunos cadastrados. O SGBD utilizado é o favorito da comunidade PHP, o MySQL, e para isso em nosso exemplo vamos utilizar o WampServer, portanto clique no ícone do WampServer que fica perto do relógio e em seguida phpMyAdmin. Dentro da ferramenta escolha o banco de dados MySQL (à esquerda da tela) e após clique no item SQL. Digite o script da **Listagem 1**.

Nota do DevMan

As partes que compõem o padrão MVC:

View

- Camada de apresentação ou visualização - Não está preocupada em como a informação foi obtida ou onde ela foi obtida, apenas exibe a informação;
- Inclui os elementos de exibição no cliente: HTML, XML, ASP, Applets;
- É a camada de interface com o usuário;
- É usada para receber a entrada de dados e apresentar o resultado.

Control

- Camada de lógica da Aplicação - É o coração da aplicação. Responsável por tudo que a aplicação vai fazer;
- Modela os dados e o comportamento por trás do processo de negócios;
- Preocupa-se apenas com o armazenamento, manipulação e geração de dados;
- É um encapsulamento de dados e de comportamento independente da apresentação.

Model

- Camada de Controle - determina o fluxo da apresentação servindo como uma camada intermediária entre a camada de apresentação e a lógica;
- Controla e mapeia as ações.

Listagem 1. Script do banco de dados

```
CREATE DATABASE academico DEFAULT CHARACTER SET latin1 COLLATE
 latin1_swedish_ci;
USE academico;

CREATE TABLE aluno (
  id_aluno int(11) NOT NULL auto_increment,
  nome varchar(400) NOT NULL,
  email varchar(255) default NULL,
  endereco varchar(400) default NULL,
  bairro varchar(150) default NULL,
  cidade varchar(100) default NULL,
  CONSTRAINT PK_ALUNO PRIMARY KEY  (id_aluno)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=27 ;
```

Classe Aluno

Vamos iniciar nosso exemplo criando o projeto orientado a objetos, para isso crie uma pasta com o nome PHP_OO dentro da pasta C:\wamp\www que é o localhost do WampServer. Dentro da pasta criada anteriormente crie o arquivo aluno.class.php conforme a **Listagem 2**. Neste momento criamos a classe aluno com seus atributos e seus respectivos métodos get e set.

Métodos privados de acesso ao banco de dados

A persistência e a consulta à base de dados será feita através de métodos privados insert_Aluno, update_Aluno, delete_Aluno e consultar_Alunos. Escreva o código da **Listagem 3** logo abaixo do método __Construct.

Observe que foram criados métodos para cada uma das ações esperadas no projeto e seus métodos são do tipo privado pelo fato de que eles devem ser solicitados por métodos que

Listagem 2. Classe Aluno

```
<?php

class Aluno {
 //atributos
 var $id_aluno;
 var $nome;
 var $email;
 var $endereco;
 var $bairro;
 var $cidade;

 //métodos get
 public function get_id_aluno()
 { return $this->id_aluno;
 }
 public function get_nome()
 { return $this->nome;
 }
 public function get_email()
 { return $this->email;
 }
 public function get_endereco()
 { return $this->endereco;
 }
 public function get_bairro()
 { return $this->bairro;
 }
 public function get_cidade()
 { return $this->cidade;
 }

 //métodos set
 public function set_id_aluno($id_aluno)
 { $this->id_aluno = $id_aluno;}
 public function set_nome($nome)
 { $this->nome = $nome;
 }
 public function set_endereco($endereco) { $this->endereco =
$endereco;
 }
 public function set_bairro($bairro)
 { $this->bairro = $bairro;
 }
 public function set_cidade($cidade)
 { $this->cidade = $cidade;
 }
 public function set_email($email)
 { $this->email = $email;
 }

 //construtor da classe
 function __Construct(){
 $this->set_nome($_REQUEST["nome"]);
 $this->set_endereco($_REQUEST["endereco"]);
 $this->set_bairro($_REQUEST["bairro"]);
 $this->set_cidade($_REQUEST["cidade"]);
 $this->set_email($_REQUEST["email"]);
 }
 }
?>
```

Listagem 3. Métodos de comunicação com o banco de dados

```
1) private function insert_Aluno($nome,$endereco,$bairro,$cidade,$email){
2) $sql = "INSERT INTO ALUNO (NOME,ENDERECO,BAIRRO,CIDADE,EMAIL) VALUES
    ('".$nome."','".$endereco."','".$bairro."','".$cidade."','".$email."')";
3) $con = mysql_connect("localhost","root","");
4) mysql_select_db("academico");
5)   $status = mysql_query($sql,$con);
6)   return $status;
7)   }

8)   private function update_Aluno($id_aluno,$nome,$endereco,
       $bairro,$cidade,$email){
9)   $sql = "UPDATE ALUNO SET NOME = '".$nome."', ENDERECO = '".
      $endereco."', BAIRRO = '".$bairro."', CIDADE = '".$cidade."',
       EMAIL = '".$email."' WHERE ID_ALUNO = '".$id_aluno."'";
10) $con = mysql_connect("localhost","root","");
11) mysql_select_db("academico");
12) $status = mysql_query($sql,$con);
13) return $status;
14) }
15) private function delete_Aluno($id_aluno){
16) $sql = "DELETE FROM ALUNO WHERE ID_ALUNO = '".$id_aluno."'";
17) $con = mysql_connect("localhost","root","");
18) mysql_select_db("academico");
19) $status = mysql_query($sql,$con);
20) return $status;
21) }
22) private function consultar_Alunos(){
23) $id_aluno = $this->get_id_aluno();
24) $sql = "SELECT ID_ALUNO, NOME, ENDERECO, BAIRRO, CIDADE, EMAIL
     FROM ALUNO";
25) $sql .= (($id_aluno!="")?(" WHERE ID_ALUNO = ".
     $id_aluno ):(""));
26) $sql .= " ORDER BY NOME";
27) $con = mysql_connect("localhost","root","");
28) mysql_select_db("academico");
29) $status = mysql_query($sql,$con);
30) return $status;
31) }
...
```

estiverem dentro da classe aluno. Em todos os métodos usamos as funções de acesso à base de dados MySQL do PHP.

A função mysql_connect() faz a conexão com o servidor MySQL, seus parâmetros são o IP do servidor, o usuário e a senha, que no caso da instalação padrão do WampServer está em branco. A função mysql_select_db() abre a conexão com o banco de dados informado e a função mysql_query() executa o código SQL passado por parâmetro juntamente com a instância da conexão com o servidor.

Métodos privados de interface

Outros métodos que devem ter o mesmo tratamento são os relacionados à interface do projeto, que são compostos por código HTML mesclados com códigos PHP. Primeiramente vamos analisar o método formCadastro() da **Listagem 4** que deve ser colocado abaixo dos métodos da **Listagem 3**.

Observe na primeira linha da **Listagem 4** que o método formCadastro tem um parâmetro e o valor esperado por ele é um array contendo uma registro da tabela Aluno. Veja nas linhas 21 e 22, 30 e 31, 39 e 40, 48 e 49, 57 e 58, e 66 e 67 que é feito um teste onde, caso haja valor no campo respectivo ao input criado, "escrevemos" a tag value e o valor retornado do array, criando assim um recurso de edição dos dados já cadastrados, ou em caso de não haver valor as caixas de texto aparecerão em branco prontas para serem preenchidas. Continuamos o exemplo inserindo logo abaixo ao método formCadastro() o método formVisualizar() que está na **Listagem 5**.

O método formVisualizar() é um método mais simples que o formCadastro(), já que somente recebe por parâmetro a lista completa dos alunos cadastrados, entre as linha 25 e 36 observe que é feito um loop com a lista de alunos utilizando a função mysql_fetch_array() e em seguida seu retorno é mesclado com as tag HTML de criação de linhas e colunas em uma tabela. A função mysql_fetch_array() lê a linha atual do array e retorna os valores a uma outra variável, no nosso caso a variável se chama linha.

Nesta tabela foram colocadas duas colunas no final da linha que servirão para realizar as solicitações de edição e exclusão de registros. Para esse exemplo foi utilizamos a URL para definir qual ação será tomada, se de edição ou exclusão. Não

Listagem 4. Formulário de cadastro

```
1)  private function formCadastro($linha) {
2)
3)  ?>
4) <table width="400" height="157" border ="3" align=center bgcolor="gray">
5)    <tr bgcolor="#AABBCC">
6) <td align="center">
7) <font color="white"><b><a href="index.php?metodo=cadastro" style="text-
    decoration: none; color:#ffffff">Cadastrar</a>
8) </td>
9) <td align="center">
10)  <font color="white"><b><a href="index.php?metodo=visualiza"
      style="text-decoration: none; color:#ffffff">Visualizar</a>
11) </td>
12) </tr>
13) <tr>
14) <td colspan="2" width="166">
15) <table width="410" height="140" border="0" bgcolor="#DDDDDD">
16)  <form method="POST" action="" name="form_cadastro">
17) <tr>
18)  <td width="51">Identificação.: </td>
19)  <td width="120" align="left"><input type="text" name="ID"
      size="10" readonly="true"
20)  <?php
21) if (!empty($linha["ID_ALUNO"]))
22) echo "value='".$linha["ID_ALUNO"]."'";
23)  ?>
24)  ></td>
25)  </tr>
26) <tr>
27)  <td width="51">Nome.: </td>
28)  <td width="120"><input type="text" name="nome" size="50"
29) <?php
30)  if (!empty($linha["NOME"]))
31)  echo "value='".$linha["NOME"]."'";
32)  ?>
33)  ></td>
34)  </tr>
35) <tr>
36) <td width="51">Endereço.: </td>
37) <td width="120"><input type="text" name="endereco" size="50"
38) <?php
39) if (!empty($linha["ENDERECO"]))
40) echo "value='".$linha["ENDERECO"]."'";
41)  ?>
42)  ></td>
43) </tr>
44) <tr>
45) <td width="51">Bairro.: </td>
46)   <td width="120"><input type="text" name="bairro" size="50"
47)   <?php
48)   if (!empty($linha["BAIRRO"]))
49)  echo "value='".$linha["BAIRRO"]."'";
50)  ?>
51)  ></td>
52) </tr>
53) <tr>
54)    <td width="51">Cidade.: </td>
55) <td width="120"><input type="text" name="cidade" size="50"
56)  <?php
57) if (!empty($linha["CIDADE"]))
58) echo "value='".$linha["CIDADE"]."'";
59) ?>
60) ></td>
61) </tr>
62) <tr>
63)    <td width="51">Email.: </td>
64) <td width="120"><input type="text" name="email" size="50"
65) <?php
66)   if (!empty($linha["EMAIL"]))
67) echo "value='".$linha["EMAIL"]."'";
68) ?>
69)   ></td>
70) </tr>
71) <tr>
72)    <td colspan="2" align="center"><input type="submit"
        value="Gravar"></td>
73) </tr>
74) </form>
75) </table>
76) </td>
77) </tr>
78) </table>
79) <?php
80) }
```

Listagem 5. Formulário Visualizar

```
1) private function formVisualizar($lista_alunos){
2) ?>

3) <table width="1000" border ="2" align=center bgcolor="gray">
4) <tr bgcolor="#AABBCC">
5) <td align="center">
6) <font color="white"><b><a href="index.php?metodo=cadastro" style="text-
   decoration: none; color:#ffffff">Cadastrar</a>
7) </td>
8) <td align="center">
9) <font color="white"><b><a href="index.php?metodo=visualiza"
   style="text-decoration: none; color:#ffffff">Visualizar</a>
10) </td>
11) </tr>

12) <tr>
13) <table width="1000" border ="2" align=center bgcolor="gray">
14) <tr bgcolor="#AABBCC">
15)  <td width="25"  align="center" style="color:#ffffff">ID</td>
16)  <td width="100" align="center" style="color:#ffffff">Nome</td>
17)  <td width="100" align="center" style="color:#ffffff">Endereço</td>
18)  <td width="75"  align="center" style="color:#ffffff">Bairro</td>
19)  <td width="50"  align="center" style="color:#ffffff">Cidade</td>
20)  <td width="25"  align="center" style="color:#ffffff">Email</td>
21)  <td></td>
22)  <td></td>
23) </tr>
24) <?php
25) while ($linha=mysql_fetch_array($lista_alunos)) { ?>
26) <tr bgcolor="#DDDDDD">
27)  <td align="center"> <?php echo $linha["ID_ALUNO"] ?></td>
28)  <td> <?php echo $linha["NOME"]     ?></td>
29)  <td> <?php echo $linha["ENDERECO"] ?></td>
30)  <td> <?php echo $linha["BAIRRO"]    ?></td>
31)  <td> <?php echo $linha["CIDADE"]   ?></td>
32)  <td> <?php echo $linha["EMAIL"]    ?></td>
33)  <td width="25"  align="center" style="color:#ffffff"><?php echo
      "<a href='index.php?metodo=cadastro&ID=".$linha["ID_ALUNO"]."'" ?>"
      style="text-decoration: none; color:blue">Editar</a></td>
34)   <td width="25"  align="center" style="color:#ffffff"><?php echo
      "<a href='index.php?metodo=visualiza&ID=".$linha["ID_ALUNO"]."'" ?>
      style="text-decoration: none; color:black">Excluir</a></td>
35) /tr> <?php
36) }
37) ?>
38) </table>
39) </tr>
40) </table>
41) <?php
42) </tr>
```

se preocupe que tudo ficará mais claro quando criarmos os métodos públicos da classe Aluno.

Métodos públicos

Os métodos públicos têm como característica as regras de negócio, ou seja, um tratamento específico para um determinado requisito funcional do sistema. Para começar vamos criar o método cad_Aluno() conforme a **Listagem 6**, insira o código logo abaixo do método formVisualizar().

O método cad_Aluno() solicita aos valores dos atributos da classe e caso pelo menos o nome e mais um outro atributo tenha valor, solicita o método insert_Aluno() que faz a persistências dos dados. Em caso contrário este verifica se esse registro já existe na base de dados e carrega a tela de cadastro com os dados encontrados.

Note que aqui é programado o comportamento do sistema, são requisitados os valores dos atributos e é feita a inserção do registro na tabela, ou a tela de cadastro é exibida.

Listagem 6. Método cad_Aluno()

```
1) public function cad_Aluno() {
2) $nome = $this->get_nome();
3) $endereco = $this->get_endereco();
4) $bairro = $this->get_bairro();
5) $cidade = $this->get_cidade();
6)  $email = $this->get_email();

7)  if (!empty($nome) && (!empty($endereco) || !empty($bairro)
     || !empty($cidade) || !empty($email))) {
8) $status = $this->insert_Aluno($nome,$endereco,$bairro,$cidade,
    $email);
9)  if ($status) {
10)  $lista_alunos = $this->consultar_Alunos();
11)  $this->formVisualizar($lista_alunos);
12)  echo "<br><center><b>Aluno Cadastrado com sucesso.</b></
      center><br>";
13)  }
14)  else
15)  echo "<br><center><b>Não foir possível inserir os dados.
      </b></center><br>";
16)  }
17)  else {
18)  $id_aluno = $this->get_id_aluno();
19)  if (!empty($id_aluno)) {
20)  $aluno = $this->consultar_Alunos();
21)  $linha=mysql_fetch_array($aluno);
22)  }
23)  $this->formCadastro($linha);
24)  }
25)  }
```

Na **Listagem 7** está o código do método atualiza_Aluno() responsável por atualizar os dados após a edição dos dados. Esse método é muito parecido com o método cad_Aluno() com a diferença de que o registro necessariamente já existe na base de dados, e ao invés de inserir um novo registro é feita a solicitação ao método update_Aluno().

A **Listagem 8** contém dois métodos, visualiza_Aluno() e exclui_Aluno(), e por sua simplicidade dispensam maiores explicações, porém são vitais para o funcionamento do projeto, já que são os métodos responsáveis por apresentar a lista de alunos cadastrados e excluir registros.

Página inicial

Criaremos finalmente o arquivo index.php que é a nossa página inicial, e é aqui que são feitas as solicitações aos métodos públicos da classe Aluno. Crie o arquivo conforme a **Listagem 9** e observe que primeiramente é feita a referência ao arquivo aluno.class.php que contém a classe Aluno, após instanciar a classe verifica-se qual método foi requisitado.

Com isso finalizamos nossa aplicação OO, o resultado será uma aplicação igual às **Figuras 6** e **7**, uma tela com um tipo de aba, vamos agora iniciar a conversão para arquitetura MVC.

Figura 6. Tela de cadastro

Listagem 7. Método atualiza_Aluno()

```
1) public function atualiza_Aluno($id) {
2) $this->set_id_aluno($id);
3) $nome = $this->get_nome();
4) $endereco = $this->get_endereco();
5) $bairro = $this->get_bairro();
6) $cidade = $this->get_cidade();
7) $email = $this->get_email();
8) if (!empty($nome) && !empty($endereco) && !empty($bairro) &&
    !empty($cidade) && !empty($email)) {
9) $status = $this->update_Aluno($id,$nome,$endereco,$bairro,
   $cidade,$email);
10)  if ($status) {
11)  $this->set_id_aluno("");
12)  $lista_alunos = $this->consultar_Alunos();
13)  $this->formVisualizar($lista_alunos);
14)  echo "<br><center><b>Dados Atualizados com sucesso.</b>
       </center><br>";
15)  }
16)  else
17)  echo "<br><center><b>Não foi possível atualizar os dados.
       </b></center><br>";
18)  }
19)  else {
20)  $id_aluno = $this->get_id_aluno();
21)  if (!empty($id_aluno)) {
22)  $aluno = $this->consultar_Alunos();
23)  $linha=mysql_fetch_array($aluno);
24)  }
25)  $this->formCadastro($linha);
26)  }
27)  }
```

Listagem 8. Métodos visualiza_Aluno() e exclui_Aluno()

```
1) public function visualiza_Aluno() {
2) $lista_alunos = $this->consultar_Alunos();
3) $this->formVisualizar($lista_alunos);
4) }
5) public function exclui_Aluno($id) {
6) $this->set_id_aluno($id);
7) $status = $this->delete_Aluno($id);
8) if ($status) {
9) $this->set_id_aluno("");
10)  $lista_alunos = $this->consultar_Alunos();
11)  $this->formVisualizar($lista_alunos);
12)  echo "<br><center><b>Registro excluído com sucesso.</b>
      </center><br>";
13)  }
14)  else
15)  echo "<br><center><b>Não foi possível realizar a exclusão.
       </b></center><br>";
16)  }
```

Listagem 9. A página inicial do projeto

```
1) <?php
   require_once("./aluno.class.php"); ?>
2) <html>
3) <body onLoad="">
4) <center>
5) <h2>Cadastro de Alunos</h2>
6) <table border=0>
7) <tr><td>
8) <?php
9) $aluno = new Aluno();
10)  if (($_REQUEST["metodo"] == "cadastro") ||
      (empty($_REQUEST["metodo"]))) {
11)  if (empty($_REQUEST["ID"])) {
12)  $aluno->cad_Aluno();
13)  }
14)  else {
15)  $aluno->atualiza_Aluno($_REQUEST["ID"]);
16)  }
17)  }
18)  else {
19)  if (empty($_REQUEST["ID"])) {
20)  $aluno->visualiza_Aluno();
21)  }
22)  else {
23)  $aluno->exclui_Aluno($_REQUEST["ID"]);
24)  }
25)  }
26)  ?>
27)  </td></tr></table>
28)  </center>
29)  </body>
30)  </html>
```

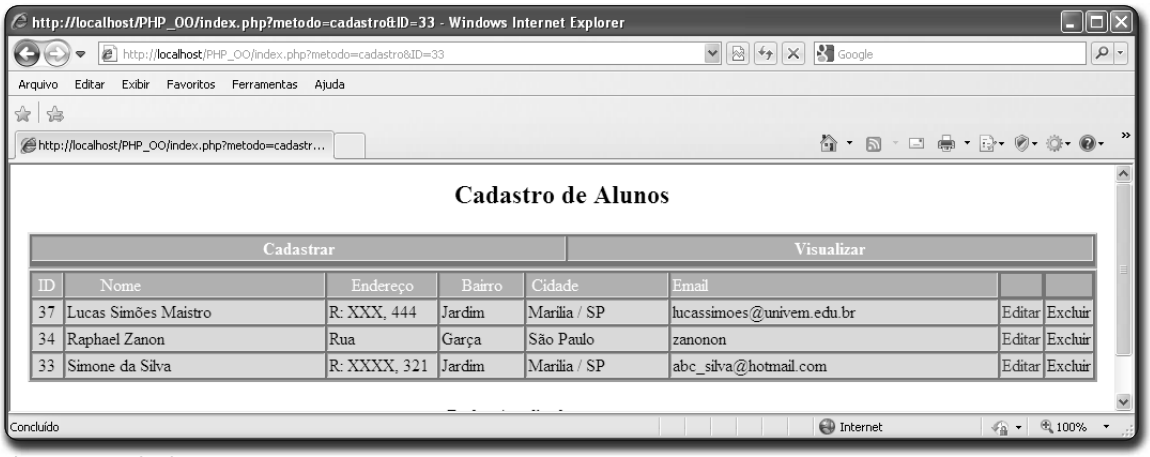


Figura 7. Lista de alunos

Nota

Para o projeto em MVC crie uma pasta chamada PHP_MVC dentro da pasta localhost do WampServer, em seguida copie o arquivo index.php do projeto anterior para a pasta atual. Vamos utilizar o mesmo arquivo pois as alterações a serem feitas para que trabalhe com as classes MVC são mínimas.

Camada Model

A camada model, como já foi mencionada, faz a comunicação com o banco de dados, portanto o código referente aos atributos, os métodos get e set e o método construtor da classe deverá ser copiado da **Listagem 2 e 3** para o arquivo aluno_model.class.php. Porém, no projeto orientado a objeto esses métodos são acessados somente de dentro da classe Aluno, já que são métodos privados, enquanto que na arquitetura MVC os métodos serão acessados por outras classes, dessa forma, é necessário que você altere a visibilidade dos métodos insert_Aluno, update_Aluno, delete_Aluno e consultar_Alunos de private para public. Seu arquivo deve ficar igual à **Listagem 10.**

Camada View

Crie um novo arquivo com o nome aluno_view.class.php, esse arquivo conterá a classe Aluno_View, responsável pelos métodos de interface da aplicação, que em nosso exemplo serão os métodos formCadastro() e formVisualizar() disponíveis nas **Listagens 4 e 5**. Veja na **Listagem 11** como fica o código da classe.

Listagem 10. Classe Aluno_Model

```
1) <?php
2) class Aluno_Model {
3) var $id_aluno;
4) var $nome;
5) var $email;
6) var $endereco;
7) var $bairro;
8) var $cidade;

9) public function get_id_aluno()
   { return $this->id_aluno;
   }
10)  public function get_nome()
     { return $this->nome;
     }
11)  public function get_email()
     { return $this->email;
     }
12)  public function get_endereco()
     { return $this->endereco;
     }
13)  public function get_bairro()
     { return $this->bairro;
     }
14)  public function get_cidade()
     { return $this->cidade;
     }
15)  public function set_id_aluno($id_aluno)
     {$this->id_aluno = $id_aluno;
     }
16)  public function set_nome($nome)
     { $this->nome = $nome;
     }
17)  public function set_endereco($endereco) { $this->endereco = $endereco;
     }
18)  public function set_bairro($bairro)
     { $this->bairro = $bairro;
     }
19)   public function set_cidade($cidade)
      { $this->cidade = $cidade;
      }
20)   public function set_email($email)
      { $this->email = $email;
      }

21) function __Construct() {
22) $this->set_nome($_REQUEST["nome"]);
23) $this->set_endereco($_REQUEST["endereco"]);
24) $this->set_bairro($_REQUEST["bairro"]);
25) $this->set_cidade($_REQUEST["cidade"]);
26) $this->set_email($_REQUEST["email"]);
27) }

28) public function insert_Aluno($nome,$endereco,$bairro,$cidade,$email){
29) $sql = "INSERT INTO ALUNO (NOME,ENDERECO,BAIRRO,CIDADE,EMAIL) VALUES
    ('".$nome."','".$endereco."','".$bairro."','".$cidade."','".$email."')";
30) $con = mysql_connect("localhost","root","");
31) mysql_select_db("academico");
32) $status = mysql_query($sql,$con);
33) return $status;
34) }

35) public function update_Aluno($id_aluno,$nome,$endereco,$bairro,$cidade,
     $email){
36) $sql = "UPDATE ALUNO SET NOME = '".$nome."', ENDERECO =
     '".$endereco."', BAIRRO = '".$bairro."', CIDADE = '".$cidade."',
      EMAIL = '".$email."' WHERE ID_ALUNO = '".$id_aluno."'";
37) $con = mysql_connect("localhost","root","");
38) mysql_select_db("academico");
39) $status = mysql_query($sql,$con);
40) return $status;
41) }
42) public function delete_Aluno($id_aluno){
43) $sql = "DELETE FROM ALUNO WHERE ID_ALUNO = '".$id_aluno."'";
44) $con = mysql_connect("localhost","root","");
45) mysql_select_db("academico");
46) $status = mysql_query($sql,$con);
```

Continuação: Listagem 10. Classe Aluno_Model

```
47) return $status;
48) }

49) public function consultar_Alunos(){
50) $id_aluno = $this->get_id_aluno();
51) $sql = "SELECT ID_ALUNO, NOME, ENDERECO, BAIRRO, CIDADE, EMAIL FROM
    ALUNO";
52) $sql .= (($id_aluno!="")?(" WHERE ID_ALUNO = ". $id_aluno ):(""));
53) $sql .= " ORDER BY NOME";
54) $con = mysql_connect("localhost","root","");
55) mysql_select_db("academico");
56) $status = mysql_query($sql,$con);
57) return $status;
58) }

59) }
60) ?>
```

Listagem 11. Classe Aluno_View

```
1) <?php
2) class Aluno_View {
3)
4) function __construct() {
   }
5)
6) public function formCadastro($linha) {  ?>
7) <table width="400" height="157" border ="3" align=center bgcolor="gray">
8) <tr bgcolor="#AABBCC">
9) <td align="center">
10) <font color="white"><b><a href="index.php?metodo=cadastro"
    style="text-decoration: none; color:#ffffff">Cadastrar</a>
11) </td>
12) <td align="center">
13) <font color="white"><b><a href="index.php?metodo=visualiza"
    style="text-decoration: none; color:#ffffff">Visualizar</a>
14) </td>
15) </tr>
16) <tr>
17) <td colspan="2" width="166">
18) <table width="410" height="140" border="0" bgcolor="#DDDDDD">
19) <form method="POST" action="" name="form_cadastro">
20) <tr>
21)  <td width="51">Identificação.: </td>
22)  <td width="120" align="left"><input type="text" name="ID" size="10"
     readonly="true"
23)   <?php
24) if (!empty($linha["ID_ALUNO"]))
25) echo "value='".$linha["ID_ALUNO"]."'";
26) ?>
27) ></td>
28) </tr>
29) <tr>
30)  <td width="51">Nome.: </td>
31)  <td width="120"><input type="text" name="nome" size="50"
32)  <?php
33) if (!empty($linha["NOME"]))
34) echo "value='".$linha["NOME"]."'";
35) ?>
36) ></td>
37) </tr>
38) <tr>
39) <td width="51">Endereço.: </td>
40) <td width="120"><input type="text" name="endereco" size="50"
41) <?php
42) if (!empty($linha["ENDERECO"]))
43) echo "value='".$linha["ENDERECO"]."'";
44)  ?>
45) ></td>
46) </tr>
47) <tr>
48) <td width="51">Bairro.: </td>
49) <td width="120"><input type="text" name="bairro" size="50"
50) <?php
51) if (!empty($linha["BAIRRO"]))
52) echo "value='".$linha["BAIRRO"]."'";
53) ?>
54) ></td>
55) </tr>
56) <tr>
57)  <td width="51">Cidade.: </td>
58) <td width="120"><input type="text" name="cidade" size="50"
59) <?php
60) if (!empty($linha["CIDADE"]))
61) echo "value='".$linha["CIDADE"]."'";
62) ?>
63) ></td>
64) </tr>
65) <tr>
66) <td width="51">Email.: </td>
67) <td width="120"><input type="text" name="email" size="50"
68) <?php
69) if (!empty($linha["EMAIL"]))
70) echo "value='".$linha["EMAIL"]."'";
71) ?>
72) ></td>
73)  </tr>
74) <tr>
75) <td colspan="2" align="center"><input type="submit" value="Gravar"></td>
76)  </tr>
77) </form>
78) </table>
79) </td>
80) </tr>
81) </table>
82) <?php
83) }
84)
85)  public function formVisualizar($lista_alunos){?>
86)  <table width="1000" border ="2" align=center bgcolor="gray">
87)  <tr bgcolor="#AABBCC">
88) <td align="center">
89)  <font color="white"><b><a href="index.php?metodo=cadastro"
     style="text-decoration: none; color:#ffffff">Cadastrar</a>
90) </td>
91) <td align="center">
92) <font color="white"><b><a href="index.php?metodo=visualiza"
    style="text-decoration: none; color:#ffffff">Visualizar</a>
93)  </td>
94) </tr>
95)  <tr>
96)  <table width="1000" border ="2" align=center bgcolor="gray">
97)    <tr bgcolor="#AABBCC">
98)  <td width="25"  align="center" style="color:#ffffff">ID</td>
99)  <td width="100" align="center" style="color:#ffffff">Nome</td>
100)  <td width="100" align="center" style="color:#ffffff">Endereço</td>
101)  <td width="75"  align="center" style="color:#ffffff">Bairro</td>
102)  <td width="50"  align="center" style="color:#ffffff">Cidade</td>
103)  <td width="25"  align="center" style="color:#ffffff">Email</td>
104)  <td></td>
105)  <td></td>
106)  </tr>
107)  <?php
108)  while ($linha=mysql_fetch_array($lista_alunos)) { ?>
109)  <tr bgcolor="#DDDDDD">
110)    <td align="center"> <?php echo $linha["ID_ALUNO"] ?></td>
111)  <td> <?php echo $linha["NOME"]     ?></td>
112)  <td> <?php echo $linha["ENDERECO"] ?></td>
113)  <td> <?php echo $linha["BAIRRO"]    ?></td>
114)  <td> <?php echo $linha["CIDADE"]   ?></td>
115)  <td> <?php echo $linha["EMAIL"]    ?></td>
116)  <td width="25"  align="center" style="color:#ffffff"><?php echo
      "<a href='index.php?metodo=cadastro&ID=".$linha["ID_ALUNO"]."'" ?>"
      style="text-decoration: none; color:blue">Editar</a></td>
117)  <td width="25"  align="center" style="color:#ffffff"><?php echo
      "<a href='index.php?metodo=visualiza&ID=".$linha["ID_ALUNO"]."'" ?>
       style="text-decoration: none; color:black">Excluir</a></td>
118)  </tr> <?php
119)  }
120)  ?>
121)  </table>
122)  </tr>
123)  </table>
124)  <?php
125)  }
126)  }
127)  ?>
```

Observe que a única mudança que ocorre é a mudança na visibilidade dos métodos que, a exemplo dos métodos da classe Aluno_Model, deixam de ser privados para serem métodos públicos.

Camada Controller

Finalmente chegamos à camada controller, nela faremos todas as solicitações aos métodos das classes Aluno_Model e Aluno_View e onde encontraremos as maiores adaptações a serem feitas. A **Listagem 12** mostra o código da classe Aluno_Controller.

Para começar devemos fazer as referências aos arquivos onde estão as classes Aluno_Model e Aluno_View conforme pode ser visto nas linhas 2 e 3. Nas linhas 6 e 7 criamos objetos mdl e vw que nas linhas 10 e 11 serão utilizados para instanciar as classes model e view.

Após o método construtor da classe Aluno_Controller devemos copiar os métodos da **Listagem 6**, porém, a exemplo da linha 15 da **Listagem 12**, todas as chamadas aos métodos das classes Aluno_Model e Aluno_View são antecedidos pela instância de suas respectivas classes (mdl-> e vw->).

Alterações na página inicial

Por fim, abra o arquivo index.php que copiamos da aplicação orientada a objetos para realizarmos as alterações necessárias. Primeiramente mude o arquivo de referência de aluno.class.php para aluno_controller.class.php. Em seguida ao invés de instanciar a classe Aluno instancie a classe Aluno_Controller e, por mais incrível que pareça, você já pode rodar sua aplicação MVC. Veja na **Listagem 13** como fica o novo código do arquivo index.php.

Conclusão

Com esse artigo o leitor passa a ter embasamento teórico e também técnico do conceito MVC, que é uma arquitetura interessantíssima, já que deixa sua aplicação mais organizada e permite maior facilidade no desenvolvimento em equipe.

A partir dos exemplos mostrados fica evidente que o MVC é algo muito mais simples do que se imagina, tanto que, com

Listagem 12. Classe Aluno_Controller

```
1. <?php
2. require_once("aluno_model.class.php");
3. require_once("aluno_view.class.php");
4.
5. class Aluno_Controller {
6.  var $mdl;
7.  var $vw;
8.
9.  public function __Construct(){
10.   $this->mdl = new Aluno_Model();
11.   $this->vw = new Aluno_View();
12. }
13.
14. public function cad_Aluno() {
15.   $nome = $this->mdl->get_nome();
16.   $endereco = $this->mdl->get_endereco();
17.   $bairro = $this->mdl->get_bairro();
18.   $cidade = $this->mdl->get_cidade();
19.   $email = $this->mdl->get_email();
20.
21.   if (!empty($nome) && !empty($endereco) && !empty($bairro) &&
       !empty($cidade) && !empty($email)) {
22.     $status = $this->mdl->insert_Aluno($nome,$endereco,$bairro,$cidade,
        $email);
23.     if ($status) {
24.     $lista_alunos = $this->mdl->consultar_Alunos();

25.     $this->vw->formVisualizar($lista_alunos);
26.       echo "<br><center><b>Aluno Cadastrado com sucesso.</b>
          </center><br>";
27.     }
28.     else
29.       echo "<br><center><b>Não foir possível inserir os dados.</b>
          </center><br>";
30.   }
31.   else {
32.     $id_aluno = $this->mdl->get_id_aluno();
33.     if (!empty($id_aluno)) {
34.     $aluno = $this->mdl->consultar_Alunos();
35.     $linha=mysql_fetch_array($aluno);
36.     }
37.     $this->vw->formCadastro($linha);
38.   }
39. }
40.
41.  public function atualiza_Aluno($id) {
42.   $this->mdl->set_id_aluno($id);
43.
44.   $nome = $this->mdl->get_nome();
45.   $endereco = $this->mdl->get_endereco();
46.   $bairro = $this->mdl->get_bairro();
47.   $cidade = $this->mdl->get_cidade();
48.   $email = $this->mdl->get_email();
49.
50.   if (!empty($nome) && !empty($endereco) && !empty($bairro) &&
      !empty($cidade) && !empty($email)) {
51.     $status = $this->mdl->update_Aluno($id,$nome,$endereco,$bairro,
        $cidade,$email);
52.     if ($status) {
53.     $this->mdl->set_id_aluno("");
54.     $lista_alunos = $this->mdl->consultar_Alunos();
55.     $this->vw->formVisualizar($lista_alunos);
56.           echo "<br><center><b>Dados Atualizados com sucesso.</b>
              </center><br>";
57.     }
58.     else
59.     echo "<br><center><b>Não foi possível atualizar os dados.</b>
        </center><br>";
60.   }
61.   else {
62.     $id_aluno = $this->mdl->get_id_aluno();
63.     if (!empty($id_aluno)) {
64.     $aluno = $this->mdl->consultar_Alunos();
65.     $linha=mysql_fetch_array($aluno);
66.     }
67.     $this->vw->formCadastro($linha);
68.   }
69. }
70.
71. public function visualiza_Aluno() {
72.   $lista_alunos = $this->mdl->consultar_Alunos();
73.   $this->vw->formVisualizar($lista_alunos);
74. }
75.
76. public function exclui_Aluno($id) {
77.   $this->mdl->set_id_aluno($id);
78.
79.   $status = $this->mdl->delete_Aluno($id);
80.   if ($status) {
81.     $this->mdl->set_id_aluno("");
82.     $lista_alunos = $this->mdl->consultar_Alunos();
83.     $this->vw->formVisualizar($lista_alunos);
84.     echo "<br><center><b>Registro excluído com sucesso.</b></center><br>";
85.   }
86.   else
87.      echo "<br><center><b>Não foi possível realizar a exclusão.</b>
        </center><br>";

88.   }
89.
90. }
91. ?>
```

Listagem 13. Página inicial

```
1) <?php
   require_once("./aluno_controller.class.php"); ?>

2) <html>
3) <body onLoad="">
4) <center>
5) <h2>Cadastro de Alunos</h2>
6) <table border=0>
7) <tr><td>
8) <?php
9) $aluno = new Aluno_Controller();
10) if (($_REQUEST["metodo"] == "cadastro") ||
    (empty($_REQUEST["metodo"]))) {
11) if (empty($_REQUEST["ID"])) {
12) $aluno->cad_Aluno();
13) }
14) else {
15) $aluno->atualiza_Aluno($_REQUEST["ID"]);
16) }
17) }
18) else {
19) if (empty($_REQUEST["ID"])) {
20) $aluno->visualiza_Aluno();

21) }
22) else {
23) $aluno->exclui_Aluno($_REQUEST["ID"]);

24) }
25) }
26) ?>
27) </td></tr></table>
28) </center>
29) </body>
30) </html>
```

o mínimo de alterações no código, transformamos uma aplicação orientada a objetos em uma aplicação MVC. Com esse exercício o desenvolvedor que já domina a POO terá extrema facilidade de compreender exatamente como funciona a arquitetura MVC, já que não foi utilizado nenhum recurso além da linguagem PHP pura. ●

Links

Site Oficial do WampServer
http://www.wampserver.com

Twitter dos Autores
http://twitter.com/lucas_simoes
http://twitter.com/zanonr

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura está acabando!

Renove Já!

www.devmedia.com.br/renovacao

MELHOR QUE SERVIDOR DEDICADO
CLOUD SERVER REDEHOST

A partir de R$ 99,00 mensais

O Cloud Server da RedeHost é a solução com melhor relação custo/benefício para aplicações que necessitem de um servidor dedicado. Por ser desenvolvido sobre conceitos de Cloud Computing utiliza da alta disponibilidade oferecida pela "nuvem computacional" com flexibilidade de expandir a capacidade de seu servidor conforme a necessidade de seu negócio.

ESCALABILIDADE

Aumente os recursos de processamento, memória, disco, e banda quando desejar.

DISPONIBILIDADE

Em caso de falha os recursos são automaticamente realocados.

PERFORMANCE

Memória, disco e banda 100% garantidos.

MENOR CUSTO

Melhor que um servidor dedicado com menor custo.

www.redehost.com.br

SP 11 4062.0909	RJ 21 4062.0909	MG 31 4062.0909
RS 51 4062.0909	SC 48 4062.0909	PR 41 4062.0909

RedeHost